Aggregating OPC UA Server for Generic Information Integration

Markus Johansson

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 19.11.2017

Thesis supervisor:

Prof. Valeriy Vyatkin

Thesis advisors:

M.Sc. (Tech.) Jouni Aro

D.Sc. (Tech.) Ilkka Seilonen

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Markus Johansson

Title: Aggregating OPC UA Server for Generic Information Integration

Date: 19.11.2017 Language: English Number of pages: 7+60

Electrical Engineering and Automation

Professorship: Control, Robotics and Autonomous Systems

Supervisor: Prof. Valeriy Vyatkin

Advisors: M.Sc. (Tech.) Jouni Aro, D.Sc. (Tech.) Ilkka Seilonen

OPC UA is an industrial communication protocol that enables the modelling of complex information with semantics and exposing it in the address space of an OPC UA server. With developments such as the Industrial Internet of Things and Industrie 4.0, the amount of data in the industrial environment is increasing and it is provided by an increasing number of sources. This can lead to information becoming increasingly scattered, which creates difficulties and inefficiencies in getting a view of all the available information.

This thesis presents the design and implementation of a software solution that can integrate information from multiple OPC UA source servers that provide information in different ways and from different viewpoints. An existing aggregating OPC UA server was improved based on elicited requirements to implement an integration platform that can group together and display the heterogeneous information sources in its specially organized address space. The developed software solution consists of three parts: instance aggregation, type aggregation and service mappings, that cooperate together to create the needed functionality.

The implemented prototype solution was evaluated in several test cases and found to meet the goals set for it. The instance aggregation procedure is able to find and group relevant information from different sources, while the type aggregation and service mappings keep the type definitions of the aggregated information intact. The instance aggregation procedure can also be configured by the user with a set of rules that enable compatibility with different use case needs. In the future, the results of this thesis will be used as a starting point in the incremental development of improved versions of the aggregation feature.

Keywords: OPC UA, aggregating server, information model

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Markus Johansson

Työn nimi: Aggregoiva OPC UA palvelin yleiseen tiedon yhdistämiseen

Päivämäärä: 19.11.2017 Kieli: Englanti Sivumäärä: 7+60

Sähkötekniikan ja automaation laitos

Professuuri: Ohjaus, robotiikka ja autonomiset järjestelmät

Työn valvoja: Prof. Valeriy Vyatkin

Työn ohjaajat: DI Jouni Aro, TkT Ilkka Seilonen

Teollisuudessa käytetty OPC UA -tiedonsiirtomäärittely mahdollistaa monimutkaisen tiedon ja semantiikan esittämisen UPC UA -palvelimen osoiteavaruudessa oliomallin avulla. Teollisen internetin ja Industrie 4.0:n viitoittama suunta teollisuudessa on lisääntyvä tiedon määrä yhä useammista tietolähteistä. Tämän seurauksena tieto voi pirstaloitua ja täten vaikeuttaa kokonaiskuvan saantia olemassaolevasta tiedosta.

Tämä diplomityö esittelee suunnittelun ja toteutuksen ohjelmistolle, joka pystyy integroimaan tietoa useista eri OPC UA -lähdepalvelimista, jotka voivat esittää tietoa eri tavoin ja eri näkökulmista. Olemassaolevaa aggregoivaa OPC UA -palvelinta kehitettiin uusiin vaatimuksiin perustuen toteuttamaan integraatioalusta, joka voi ryhmitellä yhteen ja näyttää tietoa erilaisista lähteistä tarkoituksenmukaisesti järjestetyssä nimiavaruudessaan. Kehitetty ohjelmistoratkaisu koostuu kolmesta osasta: instanssien aggregoinnista, tyyppien aggregoinnista ja palvelukartoituksista, jotka toimivat yhdessä tuottaakseen tarvittavan toiminnallisuuden.

Kehitettyä prototyyppiratkaisua arvioitiin useissa testitapauksissa ja sen havaittiin täyttävän sille asetetut tavoitteet. Instanssien aggregointi pystyy löytämään ja ryhmittelemään yhteenkuuluvat tiedot eri lähteistä, kun taas tyyppien aggregointi ja palvelukartoitukset pitävät aggregoidun tiedon tyypppimäärittelyt muuttumattomina. Käyttäjä voi konfiguroida instanssien aggregointia käyttämällä erityisiä sääntömäärittelyjä, jotka mahdollistavat aggregointiprosessin yhteensopivuuden eri käyttötarpeiden kanssa. Tulevaisuudessa tässä opinnäytetyössä saatuja tuloksia käytetään lähtökohtana aggregointitoiminnallisuuden asteittaisesssa jatkokehittämisessä.

Avainsanat: OPC UA, aggregoiva palvelin, tietomalli

Preface

I want to thank Prosys OPC for the opportunity and the resources to complete this project. My good co-workers at Prosys also helped to provide a pleasant working environment during this great endeavour.

I also want to thank my supervisor Valeriy Vyatkin and my instructors Jouni Aro and Ilkka Seilonen for their valuable feedback during the thesis process.

Espoo, 19.11.2017

Markus H. A. Johansson

Contents

Abbreviations

API	Application Programming Interface
CAEX	Computer Aided Engineering Exchange
COLLADA	COLLAborative Design Activity
CPS	Cyber-Physical System
DCS	Distributed Control System
ERP	Enterprise Resource Planning
GUI	Graphical User Interface
HMI	Human-Machine Interface
IEC	International Electrotechnical Commission
IIoT	Industrial Internet of Things
ISA	International Society of Automation
ISO	International Organization for Standardization
MES	Manufacturing Execution System
MOM	Manufacturing Operations Management
OPC	Open Platform Communications
OPC UA	OPC Unified Architecture
ORM	Object-Relational Mapping
PLC	Programmable Logic Controller
RDBMS	Relational Database Management System
SCADA	Supervisory Control and Data Acquisition
SDK	Software Development Kit
SOA	Service Oriented Architecture
SQL	Structured Query Language
UML	Unified Modeling Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language

1 Introduction

1.1 Background

The amount of data in the modern industrial environment is increasing with rising levels of digitalization and automation. The emerging concepts of Industrial Internet of Things (IIoT) and the Industrie 4.0 initiative aim to further increase the available information by introducing interconnected sensors and cyber-physical systems as providers of vast amount of data and services. The information can be used as a tool in improving decision making processes and automation control in industrial information systems, such as Enterprise Resource Planning (ERP), Manufacturing Execution Systems (MES) and Supervisory Control and Data Acquisition (SCADA). However, the key challenge regarding these developments is how to effectively manage and utilize that vast array of data that is provided by numerous decentralized providers. The first issue comes from the interoperability problems that can arise from using disparate communication protocols between various devices and systems. The second issue is related to the lack of horizontal and vertical integration that arises from having multiple independent servers instead of a centralized integration server. Accessing different systems and devices, with heterogeneous information models and from different levels of the automation pyramid, can be challenging. Configuring the required connections and security settings for multiple servers requires great effort. Furthermore, it is not easy to acquire a holistic view of information that is scattered into several servers.

The OPC Unified Architecture (OPC UA) communication protocol aims to answer these dilemmas with its interoperable nature, scalability and flexible information modelling capabilities. OPC UA enables integration of production data from the device level to different levels of information systems. Metadata and semantics provided by the information modelling capabilities of OPC UA can answer the needs of industrial information systems, in addition to basic level automation control. OPC UA can expose and access data regardless of the data content or the device platform, thus enabling interoperability between numerous systems of various applications. Currently, there are several information modelling standards for OPC UA that concentrate on modelling different aspects of systems. Different models always have certain viewpoints to the modelled systems and usually remain separate from each other due to natural separation of different engineering domains. These models might also be divided into different servers, which is often the case in the industrial environment. For example, real-time device information might be provided by automation systems while schematic design data is provided by information systems in the manufacturing operation management level. Division of data to different servers can be beneficial due to reasons of redundancy, modularity and load-balancing. However, in the worst case, this might result in several servers providing various models representing different aspects of the same entities (physical or virtual). In that case, integration of information is lost. If a user wishes to gain a complete view of the information available, he must access multiple models and possibly connect to multiple servers to combine all the existing data.

One solution to managing and accessing multiple OPC UA source servers providing heterogeneous data is the aggregating server paradigm presented in the fundamental works of Mahnke, Leitner and Damm[1] and in the OPC UA Specification[2]. The aggregating server is a single server that provides an integrated entry point to the data and services of several underlying source servers. It aims to combine the information from the source servers in its own address space in a purposeful way and can also perform transformations on the data models of the source servers if necessary. Additionally, aggregating servers can be utilized as tools of vertical integration. Aggregating servers from the lower levels of automation hierarchy can act as source servers for higher level aggregating servers. In this manner, information is organized in a way that replicates the plant structure and the higher level information systems are not required to connect to the numerous lower-level devices or systems directly.

OPC UA Historian is an existing commercial product developed by Prosys OPC Ltd. It offers functionality for collecting data values from connected servers into a database, but it also provides simple aggregating server capabilities. The aggregation functionality in the OPC UA Historian adds an entry point to each configured source server to its own address space, allowing users to browse any of the connected individual servers through these entry points. In this manner, the OPC UA Historian acts as an integration portal for multiple underlying servers. However, this functionality is very basic and keeps the information from the different servers strictly separated. Furthermore, the existing implementation introduces type consistency issues, which will be explained later in this thesis.

The aggregating OPC UA server and the integration of information from different domains in OPC UA has been researched in the Technische Hochschule Ingolstadt[3][4], in the Technische Universität Dresden[5][6], in the Tampere University of Technology[7] and in the Aalto University[9][10][11][12]. The first article by Großmann et al.[3] presents the motivation for using aggregating servers and proposes a generic architecture as well as evaluates an implemented prototype solution. The second article by Großmann and Banerjee[4] expands on the previous with respect to handling aggregation of different information models along with an overview of a possible architecture. The problem is very similar to the one presented in this thesis. However, the solution presented by Großmann and Banerjee in the article does not cover any technical or implementation details. Thus, it cannot be directly utilized in creating actual software implementations. The papers by Wollschlaeger et al.[5][6] provide an example of tackling the problem of combining heterogeneous information models from the domains of industrial and building automation by creating a new higher level homogenized information model. Hästbacka et al.[7] outline the requirements for a concept of an aggregating OPC UA server used for centralized condition monitoring of field devices and sensors. The master's theses written by Elovaara[9], Tuomi[10] and Tuovinen[11] cover topics related to aggregating servers. Elovaara studies the use of aggregating servers in agricultural work machines and Tuomi their usage in flexible manufacturing systems. Tuovinen presents a solution on how to perform address space transformations between the source servers and the aggregating server. The works are all also summarized in the conference paper by

Seilonen et al.[12].

1.2 Objectives and Scope

The basic concepts of the aggregating server in OPC UA, from the relaying of services to basic data model transformations, have been examined and proven in practice by the aforementioned research and even commercial solutions such as the UaGateway software by Unified Automation and the OPC UA Historian by Prosys OPC. Nevertheless, the issues related to aggregating several servers with various heterogeneous information models have been largely left uncovered. The central topic that this thesis is concerned with is the situation where there are various servers providing different aspects of the same entities with dissimilar information models. This problem has only been briefly discussed by Großmann and Banerjee[4].

The aim of this thesis is to first research the different information model specifications that exist for OPC UA. The purpose is to determine what are the viewpoints and specific characteristics of the information models. Secondly, this knowledge is then utilized in developing a prototype feature in the Prosys OPC UA Historian software for aggregation of different information models that display different aspects of the same entity. The goal is that the new aggregation feature is able to create a user-configurable aggregated address space where information from multiple source servers and from different information models is concentrated. Such a solution would provide an integration platform for information and an easy access point in use cases involving horizontal or vertical integration of data. The feature aims to be a generic solution and applicable in a wide array of domains, including industrial information systems. In addition, it is necessary that the solution ensures that all type definitions in the aggregated address space are consistent among instances that are integrated from different source servers. Before developing the implementation, more specific requirements need to be determined and analyzed. The final implementation is tested and evaluated using a set of simple test cases displaying different integration needs and involving various servers implementing multiple information models.

1.3 Research Methods

Primary sources for knowledge on current research on the subjects of aggregated servers and information models in this thesis were the article papers mentioned in the previous section, mainly by Großmann et al.[3][4] from the Technische Hochschule Ingolstadt and the master's theses written by Elovaara[9], Tuomi[10] and Tuovinen[11] from the Aalto University. The premier sources for the basic concepts of OPC UA communication protocol were the fundamental works of Mahnke et al.[1] and the official specification documents for OPC UA[2][13][14][15]. The official documentations of the various OPC UA companion specifications[16][17][18][19][20] were extensively studied in order to map the characteristics of the different information models.

The attained knowledge from the literature review was utilized in the software development phase. A set of requirements was specified for the new aggregation functionality and a software blueprint was designed to meet these requirements.

During implementation, the existing source code of the Prosys OPC UA Historian software was first thoroughly examined. Functional prototyping of the implementation was then used as a tool for intermediate experimentation and evaluation during application development. This was a valuable source of direction during the development process. The finished implementation was a preliminary prototype of the new features in the Historian software. The solution was then evaluated against the defined requirements and analyzed for shortcomings that could serve as areas of future development before the feature is added to the commercial version of the Historian software.

1.4 Structure of the Thesis

The work in this thesis is divided into five main chapters. This first chapter outlined the background, existing research and motivation related to this thesis and also introduced the ambitions set for the result of this work.

The second chapter introduces several important concepts related to the background of this study work. The viewpoint is the use of the OPC UA communication protocol as an integration platform as is the aim of this thesis.

The third chapter describes the characteristics of the current commercial version of the Historian software and details its issues. These issues and new demands are then used to define the requirements for the implementation that is to be developed in the course of this thesis.

The fourth chapter presents in detail the design of the implemented aggregating server feature. It covers the functionality, algorithms and architecture of the developed solution.

The fifth chapter evaluates different aspects of the implemented aggregation feature. It presents a set of test cases used to demonstrate and evaluate the functionality of the developed software. The chapter also offers a detailed analysis of different aspects of the implementation.

2 OPC UA as an Integration Platform

This chapter presents several topics that are essential background information for the topic of this thesis, namely information modelling and aggregating servers. The larger theme is the use of OPC UA communication protocol as an integration platform throughout the information systems in an industrial environment. The first section introduces the concept of information modelling on a general level and its benefits especially as a tool for integration. This is followed by a section with an overlook of the OPC UA communication protocol and its implementation of information modelling. The fourth section first presents the concept of the OPC UA companion specification and then gives a short description of some of the central companion specifications. The last section of the chapter describes the aggregating server paradigm that is a central theme of server integration in OPC UA.

2.1 Information Modelling

Information modelling is, in essence, the enrichment of basic data values with semantics and metadata that transform simple data into information. Information models define the concepts, rules, constraints, relationships and sometimes the functionality related to the information[21]. Information models usually apply to a particular limited field or a domain and different domains possess distinctive models. Information models are created to support the requirements of the specific purposes and use cases of the application domain. Hence, the scope of the model is defined already in the early stages of development[22].

Information models are usually concerned with modelling representations of general entity types and their properties, relationships and operations, but not particular instances[22]. The types can represent either abstract concepts or they can model real-world objects. Type definitions provide the constraints and the necessary properties and components needed to build a valid instance of the type. The definitions simply provide a formal way of describing entities but do not restrict the mapping of the types to existing instances.[23]

Information models also define relationships between entities. The relationships describe the internal connections and interactions between different types of instances. The classic view of information modelling incorporates three generic widely used relationships: generalization, classification and aggregation. A 'generalization' describes the relation between a superclass and a subclass, a 'classification' defines the type of an instance and an 'aggregation' depicts that an entity is comprised of other entities. However, many more relationships exist and can be utilized if they are supported by the modelling language and standards.[24] Relationships between objects is a key concept that facilitates the building of complex structures with diverse semantics that can closely relate to the characteristics of actual physical or software entities.

An important motivation behind the use of information models is that they provide structure to data and, therefore, facilitate better organization and accessibility to said data. With better organization, each user can locate and access only the information that is relevant to their demands. The well-established concept of the

automation pyramid model shows information flowing upwards through different layers of industrial information systems. The left side of the figure 1 below presents a simplified version of the automation pyramid. Basic data is generated at the machine level that is responsible for sensing and manipulating the actual physical production process. This data is collected and utilized in the control network level by systems like SCADA (supervisory control and data acquisition) and DCS (distributed control system). Further up the hierarchy, the data is utilized by systems managing the site-specific manufacturing operations, such as MES (manufacturing execution systems). At the top level, the data can be accessed through an enterprise-wide ERP (enterprise resource planning) information system. Each of these levels have their unique use cases and requirements related to the the available data. With the help of well-formed information modelling, each level of the automation hierarchy can utilize the information structures to access only the data that it requires and formulate the data to match its needs.

The emerging trend in industrial automation is towards flexible modular systems composed of cyber-physical systems (CPS) as demonstrated by strategic initiatives such as the Industrie 4.0 initiative of the German government. OPC UA communication protocol is also among the standards promoted for use in the Industrie 4.0 initiative.[25] The cyber-physical systems are autonomously exchanging information, triggering actions and controlling each other over a communication network[26]. In this way, the automation pyramid can be flattened into an Industrial Internet of Things (IIoT), where the pyramid is instead an interconnected mesh network. This structure is presented in the figure 1 below, where the left side shows the traditional layered automation pyramid and the right side shows the flattened network of IIoT. The CPS modules provide virtual descriptions of the information and services of real physical objects, which is another source of demand for sophisticated information modelling. The demands of openness and interoperability between heterogeneous system components creates the need for standardized modelling of information.

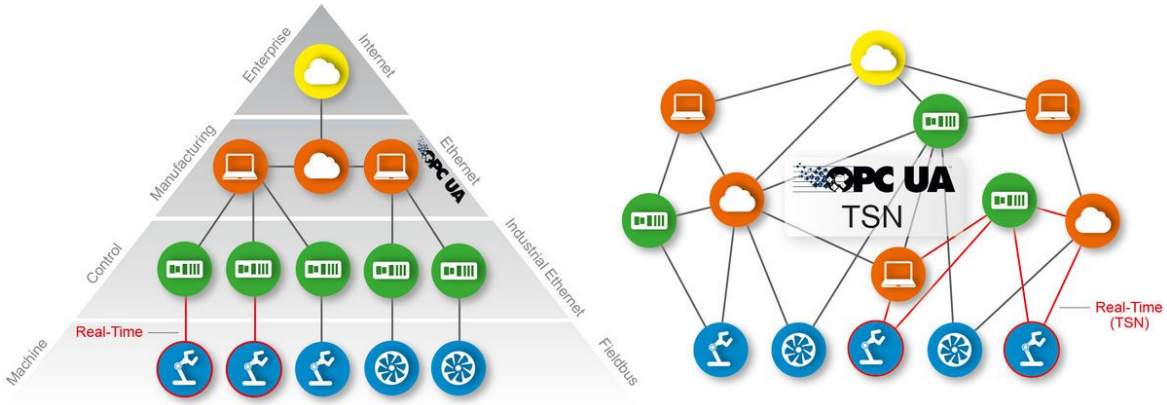

Figure 1: Structure of the automation network in a hierarchical automation pyramid (on the left) or in a mesh network of Industrial Internet of Things (on the right)[27].

However, the motivation for information modelling is not just to bring efficiency and integration, but to also provide interoperability. Without unified information

models inside common domains, different vendors and manufacturers would each organize their data in different ways. This patchwork of information would result in severe compatibility issues between different data models. The consumers of information would need to handle each vendor-specific type individually, even if they are similar in concept. On the other hand, if all the vendors model their information according to a predefined standard, it is possible for the user to handle similar types in a similar way, thus providing increased interoperability. These interoperability benefits can be maintained even when applying a vendor-specific information model, if the vendor model is derived from a common higher-level information model. Nevertheless, in order to properly use the semantics provided by an information model, the user of the information model needs to be aware of the model and be configured to utilize it.

2.2 OPC Unified Architecture

2.2.1 Introduction

Standardized in IEC 62541, OPC Unified Architecture (OPC UA) is a platform-independent communication protocol and the successor of the original OPC (also known as OPC Classic) that is widely used in industrial automation systems. Both standards have been developed by the OPC Foundation. OPC UA was released in 2008 to integrate all the different OPC Classic standards (OPC Data Access, OPC Alarms & Events and OPC Historical Data Access) under one framework. Mahnke et al.[1] summarize the improvements of the OPC UA architecture over the OPC Classic to include: platform independence, interoperability, performance, security, reliability and ability to define complex information.

The OPC UA specification defines a service-oriented architecture (SOA) with a set of services that form an interface that is utilized by OPC UA servers as suppliers of services and by OPC UA clients as consumers of services. All the possible OPC UA services are described in the part 4 of the OPC UA Specifications[14]. The services include, for example, functionalities for discovering and connecting to servers, reading and writing attribute values, exploring the address space of servers and monitoring of data changes. Part 1 of the OPC UA specifications[2] offers an overview of the OPC UA features that are described next. The OPC UA services utilize various transport protocols and data encodings for communication and data transfer between the server and client. OPC UA supports the use of transport protocols such as the internet standard HTTPS but also a faster binary OPC UA TCP protocol. Data encoding can utilize either XML or a specialized UA Binary format, however, the XML-format has not been widely adopted. The core design and definitions of OPC UA are separated from the underlying computing and network transportation technologies. OPC UA also provides a robust security model with application and user authentications and it also enables the use of encryption in the transport layer.

2.2.2 The Address Space Model

Part 3 of the OPC UA specifications[13] describes the concept of the Address Space Model that governs how information in OPC UA is modelled. The central concepts of these definitions are summarized here. Throughout this thesis, capital letters are used in representing all OPC UA–specific terms, similarly to the OPC UA specification documentation.

OPC UA employs an object-oriented data model with detailed metadata. This allows for modelling of complex systems as Objects with Variables and Methods. The Variables allow for representing data for the client and the Methods allow representing functionality. The structure of an OPC UA Object is presented in the figure 2 below.

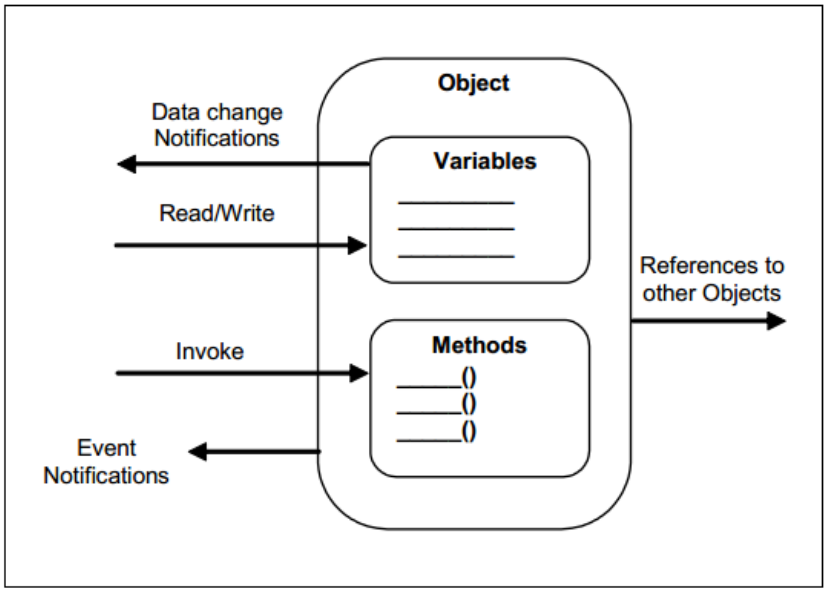

Figure 2: Structure of an Object in OPC UA[13].

The fundamental component of an OPC UA server address space is the Node. It is a structure that consists of a set Attributes and References and can be an instance of any of the eight NodeClasses (Object, ObjectType, Variable, VariableType, Method,ReferenceType, View and DataType) defined in the OPC UA specification. Each NodeClass has a set of mandatory attributes. The Node is presented in the figure 3 below. All information in the address space of an OPC UA server is constructed of these base Nodes. The Nodes have References to other Nodes that organize the information in the address space of an OPC UA server in a mesh network of Nodes. References are not strictly hierarchical but have many variations and they can be application specific.

Basic Nodes can be combined through References to define types. Types are equivalent to the class concept in programming languages. Types comprise of any number and class of Nodes and present a common structure for all instances of that type. The Address Space Model defines a set of standard ObjectTypes, DataTypes and ReferenceTypes.

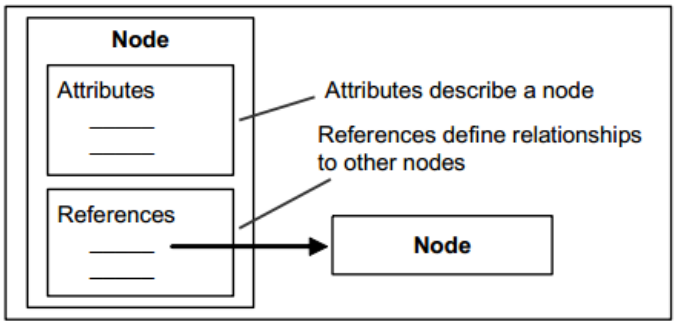

Figure 3: Structure of a Node in OPC UA[13].

2.2.3 Information Modelling in OPC UA

The OPC UA specification defines an information model as an 'organizational framework that defines, characterizes and relates information resources of a given system or set of systems'[2]. The Address Space Model is the metamodel for all information models in OPC UA[1]. All OPC UA information models utilize the concepts and restrictions of the Address Space Model to define their domain-specific formalizations.

The Part 5 of the OPC UA specifications[15] already defines a standard Information Model containing mainly basic TypeDefinitions for different NodeClasses, such as the BaseObjectType, BaseVariableType and BaseDataType as well as some of their subtypes. The base Information Model also defines the entry point for clients to the server address space and an Object named 'Server' providing diagnostic and capability information related to the server application in question. In addition, the OPC UA specifications also define some extensions to the base model for the requirements of services related to process information. These information models are divided into five separate documentations according to their use case domains: Data Access[30], Alarms and Conditions[31], Programs[32], Historical Access[33] and Aggregates[34].

Every OPC UA server contains the base Information Model, which is the model of an empty server. However, servers can support multiple information models at the same time. All other domain-specific information models extend from the base Information Model by subtyping the base types. Nonetheless, some domain-specific information models do not directly extend from the base model but instead they are derived from other domain-specific information models. In this manner, the information models in OPC UA form a layered structure where each increasingly specific model extends the more general models. This structure is shown in the figure 4 below, where the lowest level is the base Information Model. On top of the base model are the service-specific information model extensions for Data Access, Alarms & Conditions, Programs, Historical Access and Aggregates. Above this composition of general information models defined in the OPC UA specifications are the companion specifications. Companion specifications are domain-specific information models that are specified jointly by organizations operating in those

domains. The concept of companion specifications and details of some of the central companion specifications are covered in more detail in the next section of this chapter. On the uppermost level of the OPC UA information model structure are the highly specified information models defined by different companies or vendors for use in their specific products. The composition of information models can be extended even further, as each server application may specify more detailed information models specializing even further from the vendor-defined information models.

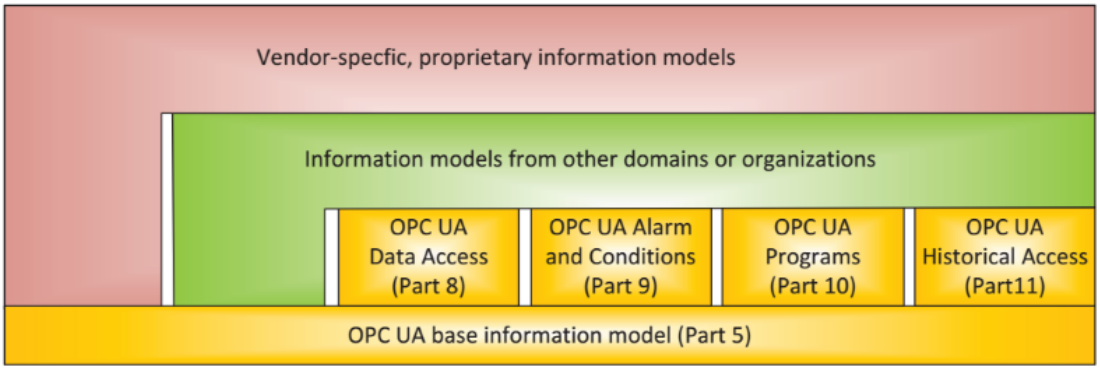

Figure 4: The layered structure of information models in OPC UA[35].

The server communicates the information model it utilizes in its address space in the form of definitions of Nodes. This type information can be read from the address space and utilized by any connected client. The Nodes defined by information models can be of any NodeClass. Type definitions are communicated in the form of ObjectTypes, VariableTypes, DataTypes and ReferenceTypes. Besides them, the information model can also define well-defined instances such as Objects and their standard Properties and Methods and even new ModellingRules. In addition, any kind of constraints can be defined, such as semantic (e.g. naming conventions) and DataType restrictions.

Each Node in the OPC UA address space has a NodeId Attribute that acts as a server-wide identifier for the Node. The NodeId contains a NamespaceIndex or a NamespaceUri. The NamespaceIndex is merely an index used to access a NamespaceUri from the NamespaceArray of the server. Therefore both the NamespaceIndex and the NamespaceUri correspond to a URI value that identifies the naming authority for the Node. This URI value is used to state the information model that the Node belongs to. Thus, each Node is uniquely tied to its information model and similar Nodes from different information models can be distinguished from each other. The NodeId also contains a value part that acts as an unique identifier in the scope of the information model marked by the URI.

OPC UA provides an object-oriented approach to information modelling. Objects, in OPC UA terms, are composed of other Objects, Variables and Methods. Objects and their components all fall under the term Node and therefore possess sets of Attributes and References. The relationships between entities in an OPC UA information model are expressed by utilizing the References. The Node that contains the Reference is the source Node and the Node that is referenced is the target Node.

References can be hierarchical and signal aggregation (e.g. 'is component of') or be non-hierarchical and signal some other association (e.g. 'is caused by'). Additionally, References have the characteristic of being either non-symmetric, i.e. directed (as in the previous examples), or they might be symmetric (e.g. electrical connection). References can also be cyclical, which means that each Node can have a Reference to any another Node. This indicates that the address space is structured as a net instead of a tree. Due to the extensible nature of OPC UA information models, it is possible to define References that indicate any possible relationship. With the use of Nodes as basic building blocks and References as the providers of semantic information, OPC UA facilitates the construction of complex objects that can model all sorts of semantic structures. The flexible information modelling capabilities provide for the usage of OPC UA in communication across the entire information network of a production plant, starting from the plant floor network (e.g. sensors and actuators), moving to the operations network (e.g. MES) and up to the corporate network (e.g. ERP).

2.3 Companion Specifications

The OPC UA companion specifications aim to provide standards for presenting information and the semantics related to a certain application domain. They are designed by organizations and industry groups from a certain field to 'define how their specific information models are to be represented in OPC UA Server AddressSpace' [2]. Each companion specification provides an information model that can be utilized in a domain or industry field that can range from very specific to very broad. An example of a companion specification with a rather specific domain is the OPC Unified Architecture for AutoID, which defines an information model for representing and accessing AutoID devices in the field of automatic identification and data capture. An example of a broader information model is the OPC Unified Architecture for Devices, which allows for representing generic devices in OPC UA. By utilizing these domain-specific information models in the design and development of new applications, it is possible to increase the interoperability between different applications of the same domain due to the similar conventions and semantics. Typically in many industrial information systems, each level of the automation pyramid is strictly separated with their own functional requirements and communication standards. This has lead to dehabilitating interoperability issues between the layers. The companion specifications built upon the OPC UA base Information Model aim to alleviate these problems in vertical integration of automation systems.

The following subsections briefly introduce a few prominent OPC UA companion specifications. The presented specifications were chosen because they were regarded as having high estimated potential as being widely used in the industrial environment and other application domains. For a product that aims for general usage and wide applicability, evaluating the most common domains is the most beneficial. All of the chosen specifications also have released versions and no release candidate or draft versions were evaluated. The presented companion specifications were used in the development and evaluation of the solution presented as the result of this thesis.

The following descriptions introduce some of the central elements of the companion specifications but are by no means complete descriptions. The reader is advised to resort to the respective specification documentations for more details. In the descriptions, it should be noted that all the type definitions have the suffix 'Type' in their name and their respective instances are without the suffix.

2.3.1 OPC Unified Architecture for Devices

The OPC Unified Architecture for Devices, or briefly Device Integration (DI), is a central information model expanding on the standard OPC UA information model with basic concepts for describing general aspects of devices. In this subsection, a description of the characteristics of the DI information model is presented based on the official documentation[16]. The specification defines a device as an 'independent physical entity capable of performing one or more specified functions in a particular context and delimited by its interfaces'. Typical devices provide sensing, actuating, communication and possibly control functionality. Examples include transmitters, valve controllers, drives, motor controllers, programmable logic controllers (PLC), and communication gateways. The objective of the specification is to unify the approach to describing automation devices in the OPC UA address space, irrespective of their underlying technological details. The Device Integration information model is also extended upon by other companion specifications, such as OPC Unified Architecture for Analyser Devices, OPC UA Information Model for IEC 61131-3 and some unreleased information models such as the companion specification for Field Device Integration (FDI).

The DI specification describes devices with three different models: the base Device Model, the Device Communication Model and the Device Integration Host Model. The Device Model provides an unified view of devices, independently from any related protocols. The Device Communication Model, on the other hand, describes the network and communication aspects of the device. Lastly, the Device Integration Host Model adds additional elements and rules that reflect the topology of the automation systems and the communication networks containing the devices. It provides information for managing the integration of the devices into a complete system. The DI companion specification mainly defines ObjectTypes with their Properties and Methods but also some Object instances as the entry points in the address space for information modelled according to the specification.

The key elements of the Device Model are the type definitions for TopologyElementType, ConfigurableObjectType and ProtocolType. An overview of the Device Model is displayed in the figure 5 below. TopologyElementType is the base ObjectType used to model elements of a device topology and it includes a set of parameters and methods for the element. The TopologyElementType is further inherited by DeviceType and BlockType. The DeviceType is an abstract type used to model general devices and should be subtyped by more specific models from vendors. It also dictates some Properties that are common to all instances, such as 'SerialNumber', 'Model' and 'Manufacturer'. BlockTypes are typically used as means to organize the functionality within a Device and the purpose of the ConfigurableObjectType is to

model modular TopologyElements that can, for example, contain various Blocks. In addition, the Device Model defines a FolderType Object called 'DeviceSet' inside the 'Objects' folder defined in the OPC UA base information model. The 'DeviceSet' aggregates all the instances of the devices implementing the information model. For hierarchical devices, only the top-level element is referenced inside the 'DeviceSet' folder.

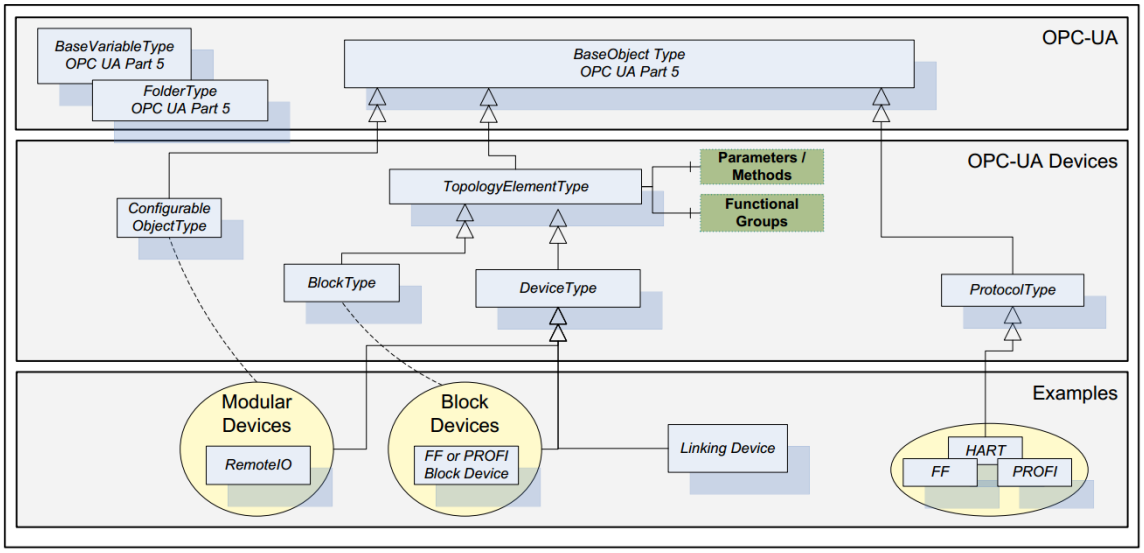

Figure 5: Overview of the Device Model of the DI information model[16].

The Communication Model defines further ObjectTypes for representing the network aspects of devices. NetworkType models the means of communication used by the Devices connected to it, for example wired or wireless technologies. ProtocolType is used to represent the communication protocols supported by TopologyElements. ConnectionPointType is a subtype of the TopologyElementType that represents the interface of a Device to a communication network. The Communication Model additionally defines an Object instance named 'NetworkSet' inside the 'Objects' folder that contains all Network instances. Communication semantics are further enhanced with two new implicit ReferenceTypes: ConnectsTo and ConnectsToParent.

The Device Integration Host Model defines an entry point named 'DeviceTopology' inside the 'Objects' folder as the starting point used to organize and provide access to all instances that constitute the device topology, such as networks, devices and communication elements. 'DeviceTopology' also contains a Property showing whether the server is currently able to communicate to Devices in the topology.

2.3.2 OPC Unified Architecture for Analyser Devices

The OPC Unified Architecture for Analyser Devices, or briefly Analyser Device Integration (ADI), specifies an unified information model to characterize different kinds of analytical devices. This section summarizes the aspects of the companion specification presented in the official documentation[17]. The analysers are divided

into various groups such as light spectrometers, particle size monitoring systems, imaging particle size monitoring systems, acoustic spectrometers, mass spectrometers, chromatographs, imaging systems and nuclear magnetic resonance spectrometers. The ADI information model is a specialization of the Device Integration information model, which is extended by the ADI model through subtyping. The information model mainly defines ObjectTypes but also a few ReferenceTypes, VariableTypes and DataTypes.

The modelling of an analyser object in the specification is mainly divided into five different definitions: AnalyserDeviceType, AnalyserChannelType, StreamType, AccessoryType and AccessorySlotType. They all inherit from types defined in the DI information model. An overview of the ADI information model as well as its connection points to the DI information model are displayed in the figure 6 below. AnalyserDeviceType is a subtype of the DeviceType of the Device Integration information model and it represents the analyser instrument as a whole. AnalyserChannelType, StreamType and AccessoryType are subtypes of the TopologyElementType and AccessorySlotType is a subtype of the ConfigurableObjectType. Each AnalyserDevice instance has at least one AnalyserChannel and may have AccessorySlots through which an Accessory can be connected. In addition, each AnalyserChannel may have its own AccessorySlots through which Accessories are connected. Accessories can only be connected to the AccessorySlots. Data acquisition is performed through the AnalyserChannel or through the Accessory connected to that AnalyserChannel. Stream is the connection of an AnalyserChannel to a specific sampling point in the monitored process. In addition, the specification defines that each of the mentioned main elements also contains a state-machine describing the current status of the device and the information model offers some new subtypes of FiniteStateMachineType from the base OPC UA Information Model to facilitate these uses.

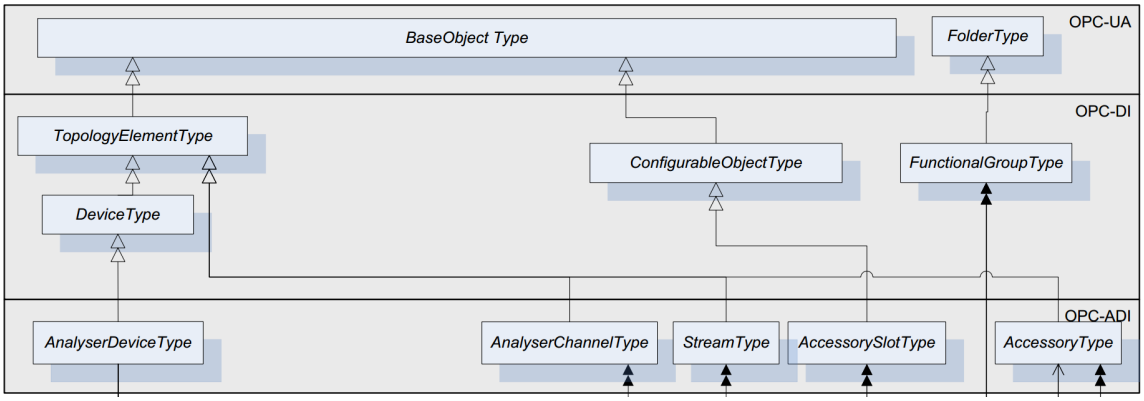

Figure 6: Overview of the Analyser Device Integration information model[17].

2.3.3 OPC UA Information Model for IEC 61131-3

IEC 61131-3 is a global vendor-independent standardization for programming languages in industrial automation. The specification of the OPC UA information

model for IEC 61131-3, often referenced simply as PLCopen, presents the types and conventions needed to make relevant information of an IEC 61131-3 compliant PLC accessible via OPC UA. An overview of the PLCopen information model is presented here based on the official specification[19]. The specification defines four use cases for the information model: observation, operation, engineering and service. Each use case extends the former. This means that, for example, the operation use case incorporates all the actions of the observation use case abut also adds more. The most basic 'observation' use case only includes the reading and monitoring of data. The 'operation' use case adds the writing of data, such as variable values, and execution control. The 'engineering' use case extends the former use cases with the writing of program elements, for example, when updating a program. The most advanced 'service' use case comprises of performing functions related to service and maintenance, such as reading or writing of special data and firmware updates.

The IEC 61131-3 information model is also a specialization and extension of the DI information model. By reason of the IEC 61131-3 specializing in the domain of controller devices, it is natural that the ObjectType definitions are subtypes of the types in the Device Integration information model. The definition for 'controller' provided by the specification is that 'a controller is a digitally operating electronic system, designed for use in an industrial environment, which uses a programmable memory for the internal storage of user-oriented instructions for implementing specific functions such as logic, sequencing, timing, counting and arithmetic, to control, through digital or analogue inputs and outputs, various types of machines or processes'. The term 'controller' is abbreviated as 'Ctrl' in the specification so as to not overlap with terms defined in the main specifications for OPC UA. The information model specification establishes a set of ObjectTypes and some ReferenceTypes.

The main concepts of the IEC 61131-3 standard and of the related OPC UA companion specification are presented here. CtrlVariables present the variables used in the control tasks and can be assigned a role of input, output or internal. They also have associated data types and scopes (local or external). The CtrlVariables implement the BaseDataVariableType of the OPC UA base Information Model. CtrlConfigurations, CtrlResources and CtrlTasks are elements used to organize the topology of a control solution. They are specified in the information model by their corresponding ObjectTypes named CtrlConfigurationType, CtrlResourceType and CtrlTaskType. CtrlConfigurationType and CtrlResourceType are subtypes of DeviceType from the DI information model and CtrlTaskType is a subtype of BaseObjectType of the base Information Model. Overview of the PLCopen information model is presented in the figure 7 below. CtrlConfiguration is the highest level entity that models the entire software needed to solve some control application. It contains at least one CtrlResource which is a component able to execute the control software. A CtrlResource can include one or more CtrlTasks that define the execution of the control software and can run periodically or triggered by events.

The actual software run by CtrlTasks can be written in any of the programming languages in the IEC 61131-3 standard and exposed in the OPC UA address space using the types derived from CtrlProgramOrganizationUnitType, which is an abstract subtype of the BlockType from the DI information model. CtrlPro-

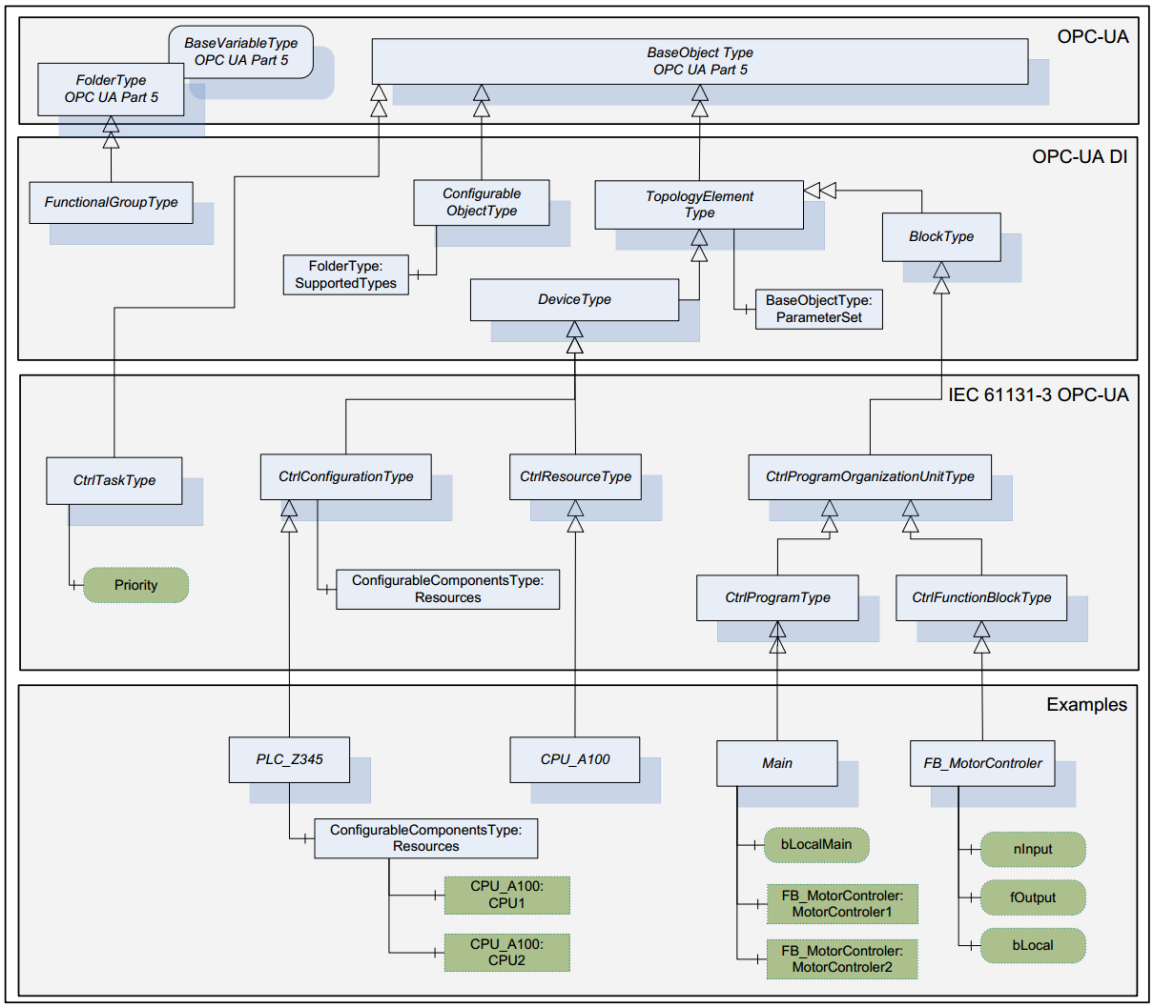

Figure 7: Overview of the PLCopen information model[19].

gramOrganizationUnitType dictates how to model the variables and the code of the element and it has two more specialized abstract subtypes CtrlProgramType and CtrlFunctionBlockType. Since all these types are abstract, no instances of them will exist and all vendors and applications must define their implementations as new subtypes.

2.3.4 OPC Unified Architecture for ISA-95 Common Object Model

The OPC Unified Architecture for ISA-95 brings the ability to model information conforming to the Common Object Model of the ISA-95 standard[36]. The ISA-95 standard is developed by the International Society of Automation to define the information interface between control systems and other enterprise systems in the industrial environment. This section will give a short introduction to the ISA-95 standard and then present the basic concepts of the information model for OPC UA based on the official specification documentation[18]. The ISA-95 standard divides the automation hierarchy into five levels, where the first level (level 0) is the actual

physical production process. The second level (level 1) consists of the devices sensing and manipulating said process, while the third level (level 2) contains automation systems controlling the process, for example, PLCs and DCSs. The fourth level (level 3) consists of manufacturing operations management (MOM) that includes various different functions such as SCADA, maintenance, quality assurance and inventory management. The final level includes business planning and logistics functions, such as ERP systems. The ISA-95 information model for OPC UA is mainly concerned with level 4 functions, i.e. MOM, and its internal communication, but also with the communication between levels 4 and 5.

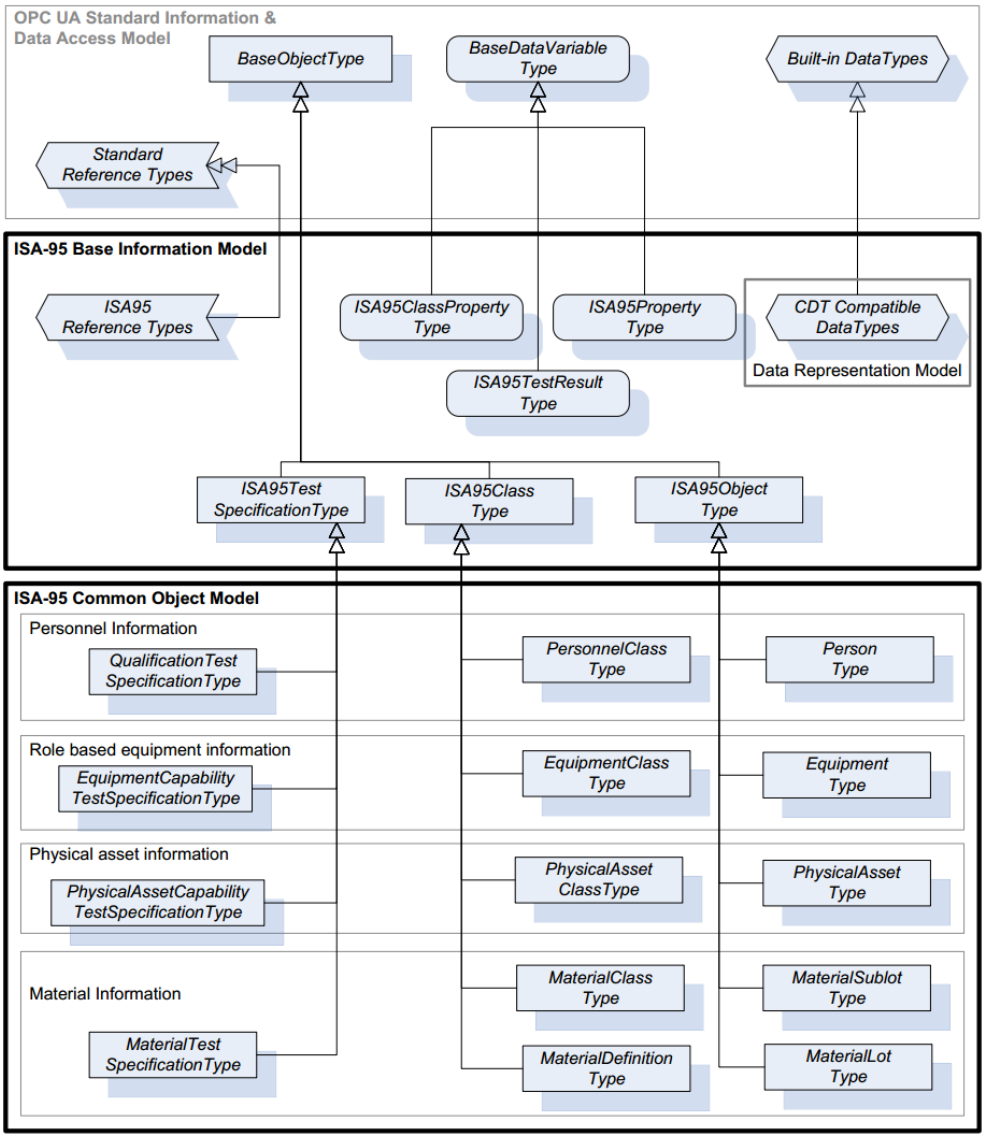

Figure 8: Overview of the ISA-95 information model[18].

The ISA-95 Common Object Model divides the information on the resources of manufacturing operations and control into four categories: personnel, role-based equipment, physical assets and material. Personnel information includes information

about personnel and their roles and qualifications. Role-based equipment information describes equipment as it pertains to the operations being performed. Physical asset information identifies the actual devices that make up the equipment. Material information defines the material and the properties of materials used in the production.

In the ISA-95 resource model, there are classes and instances. The class in ISA-95 does not necessarily mean the same as in the typical object-oriented meaning, but a classification (such as a personnel classification). Each class can have their own class-specific properties. These properties may also possess related test specifications. The classes are then implemented by instances that have values for the specified properties.

The ISA-95 Common Object Model specification for information modelling in OPC UA contains a base model and then additional models for each of the four resource categories. An overview of the structure is depicted in the figure 8. The ISA-95 base model defines three base ObjectTypes, named ISA95ClassType, ISA95TestSpecificationType and ISA95ObjectType. They relate to the concepts of class, test specification and instance that were presented in the previous paragraph. In addition, the ISA-95 base model defines two VariableTypes named ISA95ClassProperty and ISA95Property that relate to the class and instance properties. Finally, the base model defines a few new DataTypes and several ReferenceTypes.

The ISA-95 base model is further expanded by models for each of the resource categories: personnel, equipment, physical asset and material. Each of them define more VariableTypes, ReferenceTypes and ObjectTypes that are subtypes of the types in the base model. The main characteristic of the extensions is that they each specify class, instance, property and test specification types that are specific to their respective resource categories. For instance, the equipment model defines the types named EquipmentClassType, EquipmentType, EquipmentClassPropertyType, EquipmentPropertyType and EquipmentCapabilityTestSpecificationType.

2.3.5 OPC UA Information Model for AutomationML

The OPC UA information model for AutomationML enables the presentation of AutomationML information and files in the address space of an OPC UA server. The basic concepts of the AutomationML information model are summarized here on the basis of the official specification [20]. AutomationML is an open and technology-neutral standard for storage and exchange of engineering data on production systems. The standard endeavours to interconnect the different heterogeneous engineering tools used in different disciplines such as plant planning, mechanical engineering, electrical engineering, process engineering, process control engineering, human-machine interface (HMI) development, PLC programming and robot programming.

The AutomationML standard is based on XML and the top-level format is CAEX (Computer Aided Engineering Exchange). However, the standard has a modular format that includes two other XML-based data formats, COLLADA and PLCopen XML, and it also facilitates other extensions. Information is modelled according to the object-oriented paradigm which enables the modelling of physical and logical system components as data objects with various characteristics. Objects can additionally be

comprised of other objects.

Typical objects in plant automation comprise information on topology, geometry, kinematics, logic and behaviour. This information is modelled in AutomationML using an instance hierarchy. Instance hierarchy is the main model that represents the structure of the modelled system as a hierarchical tree of elements. The structure and semantics of the elements are defined by three element types: system unit classes, role classes and interface classes. The instance hierarchy tree is constructed from internal elements referencing the system unit classes they are derived from, the role classes defining their semantics and the interface objects that link objects among each other or with externally modelled information (such as external files). Definitions of the system unit, role and interface classes are divided into their respective libraries.

The AutomationML information model for OPC UA is divided into two parts: AutomationML Base Types OPC UA Model and AutomationML Libraries OPC UA Model. Both models define a set of ObjectTypes and several Object instances as entry points to information implementing the specification. In addition, the Base Types OPC UA Model defines two ReferenceTypes and a VariableType.

The Base Types Model defines the basic building blocks of AutomationML information through the CAEXBasicObjectType and its subtypes. CAEXBasicObjectType is a subtype of BaseObjectType of the base OPC UA Information Model and it describes all general characteristics of a CAEX–element in an AutomationML model. It has two subtypes called CAEXFileType and CAEXObjectType that define further features for representing CAEX files and objects. The CAEXObjectType is inherited by three ObjectTypes named AutomationMLBaseInterface, AutomationMLBaseRole and AutomationMLBaseSystemUnit. These type definitions describe the characteristics for the previously explained interface, role and system unit classes. All additional subtypes used by vendors should expand on these types. Furthermore, the Base Types Model defines instances in the OPC UA address space for organizing the data modelled using the specification. Inside the Objects folder of the OPC UA base model, the AutomationML specification adds three new folders called AutomationMLFiles, AutomationMLInstanceHierarchies and AutomationMLLibraries. The AutomationMLFiles folder is the entry point for browsing all instances of AutomationML files, the AutomationMLInstanceHierarchies contains all instance hierarchies and the AutomationMLLibraries folder presents an entry point for Browsing when looking for AutomationML libraries. The AutomationMLLibraries is further divided into three folders: InterfaceClassLibs, RoleClassLibs and SystemUnitClassLibs.

The Libraries Model populates the subfolders of AutomationMLLibraries with sets of ObjectTypes representing different interface, role and system unit classes. Vendors can utilize these predefined types in their application or supplement them with new types.

2.4 The Aggregating Server Pattern in OPC UA

The aggregating server pattern is a server architecture for OPC UA that was proposed by Mahnke et al.[1] and has since been employed in many research projects and even commercial products, such as the Prosys OPC UA Historian. An aggregating OPC

UA server encompasses both server and client implementations. The embedded client is used to access a number of other servers, called source servers or aggregated servers (the first term is used in this thesis). A typical structure of an aggregating server is presented in the figure 9. The function of the aggregating server is to concentrate information from one or more source servers in its own address space. The great benefit of an aggregated server is that it offers a central access point to multiple sources of information. A client wishing to satisfy its information needs must only connect to one central server instead of making separate connections and service requests to each of the source servers. In this way, aggregating servers are naturally suited for supervision and monitoring tasks, such as in SCADA systems, especially if the aggregating server also implements the Alarms & Conditions services of OPC UA. In addition, an aggregating server can act as a centralized security manager for connection control and greatly ease the processes of managing and supervising security[3]. The information from the source servers can be straightforwardly copied to the address space of the aggregating server or it can be transformed according to some predefined rules. The information models of the underlying servers can vary greatly and offering an unified view of several servers might require some modification to the information.

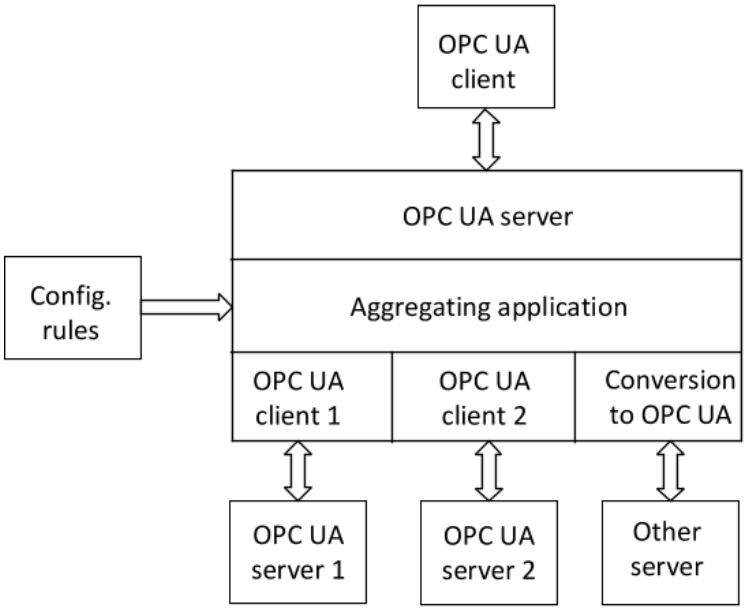

Figure 9: Model of an aggregating OPC UA server[12].

In addition to offering an integration point for information, the aggregating server is also responsible for relaying service calls to the underlying servers. The aggregating server acts as a gateway to the underlying servers and all service calls (for example reads, writes and subscriptions) must be directed to the source servers to access the actual data and functionality. The relaying of service calls is performed using the internal clients of the aggregating server. In practical implementations, the aggregating server usually has multiple internal clients and each client is responsible for communication with one source server. The passing of the service calls requires

that each Node in the aggregated address space has a counterpart Node in the address space of one of the source servers. The correspondence between Nodes must be configured in the aggregating server. However, it is also possible that the value of a Node in the address space of an aggregated server is a function of the values of multiple source Nodes. Service calls made to these aggregate Nodes might necessitate passing calls to multiple underlying servers.

Depending on the server, some of the services can be implemented by the aggregating server instead of the source servers. For example, the history data for the values of variables can be recorded by the aggregating server, which is the method used in the Prosys OPC UA Historian. In the case of the Prosys OPC UA Historian, service requests to read history data are implemented inside the aggregating server and not relayed to the underlying servers. In this manner, it is possible to implement additional services inside the aggregating server, even if the underlying source servers do not provide them. All of the services can be implemented by the aggregating or the source server or the implementations can be divided in various ways. It is also possible that a service functionality is implemented jointly by both the source server and the aggregating server in a hybrid implementation. Nevertheless, even if the aggregating server is responsible for implementing some of the services, the underlying real-time data and server-specific functionality must at some point be accessed from the source servers. Sometimes concentrating the needed services to the aggregating server can be beneficial to ensure identical and repeatable behaviour.

From the viewpoint of a client, an aggregated server functions similarly to any other server. Furthermore, aggregating servers can be aggregated by other aggregating servers to form a multi-level hierarchical server architecture. Such an architecture can mirror the structure of an automation pyramid. Higher-level aggregating servers consolidate and organize data from lower-lever servers and possibly transform the information to a more suitable form, for instance, creating abstractions required by higher levels. In this manner, each level of the automation hierarchy can have its own aggregating server that is then aggregated by higher ones to achieve vertical integration from device level up to enterprise level. A schematic of this kind of a chained aggregating server architecture is presented in the figure 10. However, a limiting factor in the usage of chained aggregating servers are the delays induced by the relaying of service requests and responses. Especially in architectures with several levels, the transmitting of requests and the respective responses through all the servers in the hierarchy can generate significant delays compared to direct communication.

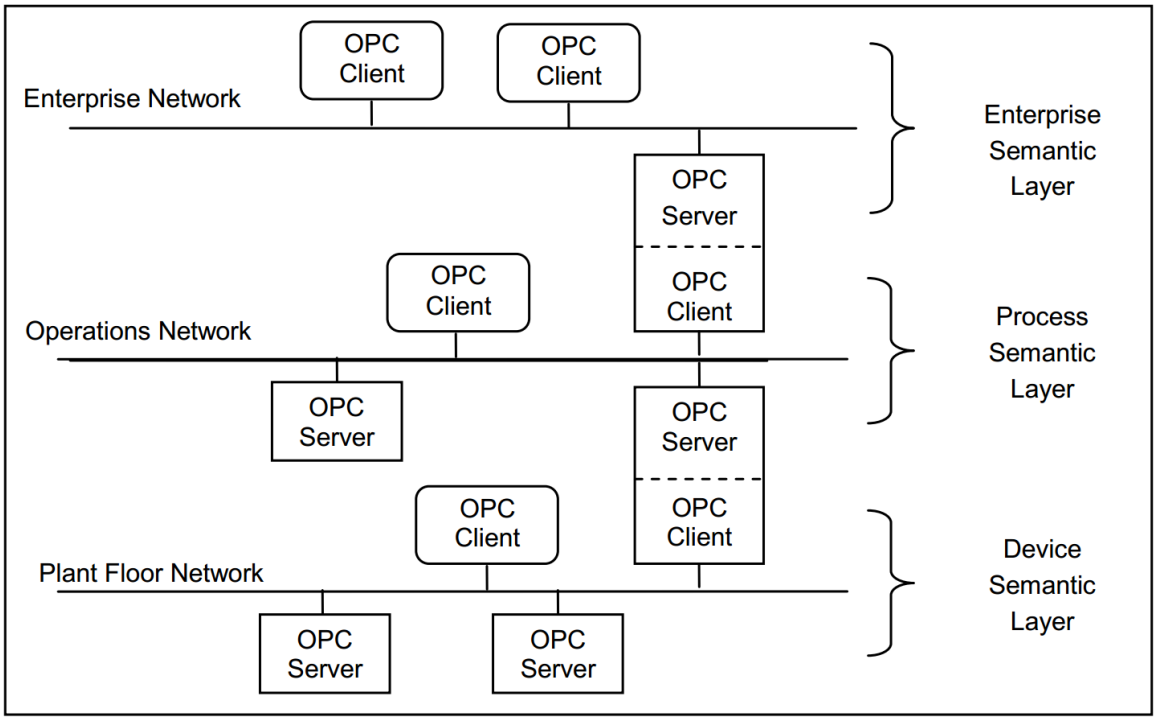

Figure 10: Schematic of a chained aggregating server architecture for vertical integration in industrial information systems[2].

3 Current Situation & New Requirements

In this chapter, the companion specifications presented in the previous chapter are first analyzed to draw conclusions that can aid in developing the new aggregation feature. Secondly, the current state of the Prosys OPC UA Historian software and its key components are presented. These are relevant to this thesis because any additional functionality will have to be built on existing software and utilizing existing software architecture. Finally, current issues related to the Historian are discussed along with new demands for the functionality. This analysis is then used to formulate the requirements for the software feature implemented in the course of this thesis.

3.1 Analysis of Companion Specifications

The different companion specifications presented earlier in this thesis, namely the information models for devices, analyser devices, IEC 61131-3, ISA-95 and AutomationML, were compared and analyzed. These standardized information models are designed to have widespread use in different applications and therefore studying their features, application domains and possible overlappings could be beneficial. These insights can be utilized in the development of the aggregation feature in the practical part of this thesis.

The mentioned companion specifications are all applied to modelling the systems and equipment in an industrial environment. Additionally, the ISA-95 is also used to represent materials and personnel. The aspects that are modelled using the companion specifications include the properties and parameters as well as the topology and composition of the modelled elements. The PLCopen and ADI information models are specializations of the DI information model. The PLCopen information model applies to modelling the aspects of software in devices, such as the software topology and parameters. The ADI specification extends the DI model by adding more type definitions that are specific to analyser devices, but it also adds concepts for modelling the state-machine of the device.

Even though all the companion specifications are used to model industrial systems and equipment, this information can be divided into two viewpoints: physical assets and equipment. The same categorization is used by the ISA-95 standard. Role-based equipment describes a device that performs some function regardless of the technical specifics. On the other hand, physical assets are the actual devices and systems that perform the functions. Physical assets have defined technical attributes and vendor-related properties, such as a model number. Role-based equipment is modelled by the ISA-95 and AutomationML specifications and physical assets by the ISA-95 and DI models. Therefore, the only overlappings are between the ISA-95 and the DI model regarding physical assets and between ISA-95 and AutomationML regarding equipment. In these cases, it is possible that different information models present the same information from the same viewpoint. It can be concluded that the information models have mostly quite distinctive application domains and do not greatly overlap with each other. The figure 11 below summarizes the observations presented in this section. The colours signify different categories of information: physical assets,

equipment, material and personnel. The ellipses describe the aspects of modelled entities present in the different companion specifications.

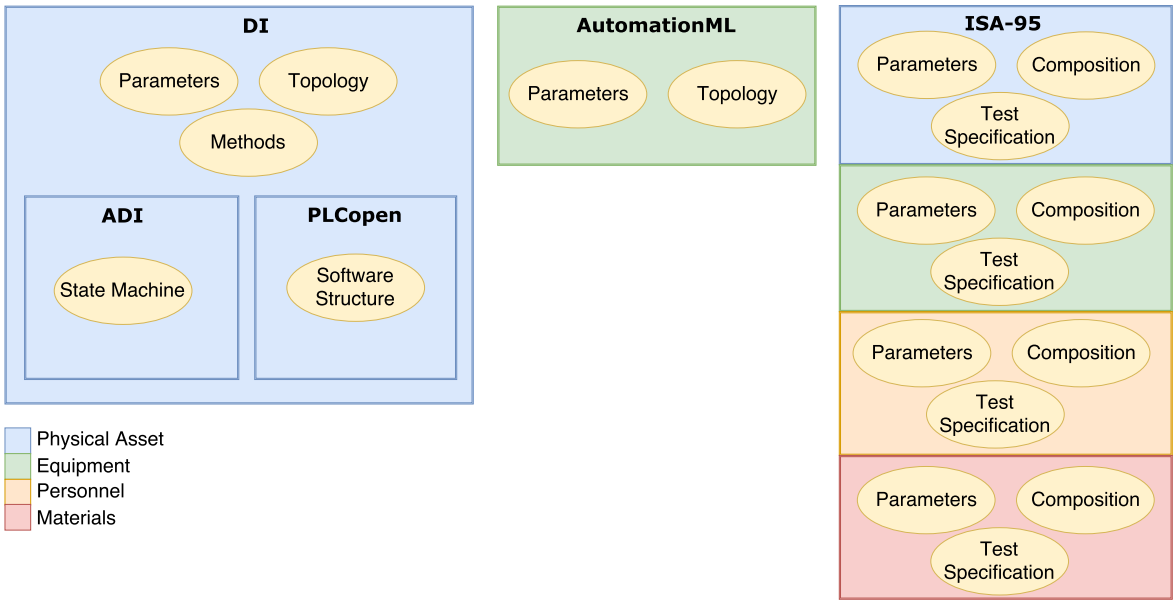

Figure 11: Overview of the features modelled by the analyzed companion specifications.

3.2 Prosys OPC UA Historian

3.2.1 Introduction

The company Prosys OPC Ltd., where this thesis work is carried out, already has a software product incorporating a simple aggregating OPC UA server. This application is called the Prosys OPC UA Historian or Historian for short. The Historian is programmed using the Java programming language and utilizes the Prosys OPC UA Java SDK in its implementation of OPC UA functionality. Primarily the Historian is a data logging software for OPC UA. The user can configure source servers to which the Historian connects and the values that the Historian collects to a SQL database. The data logged into the database can be accessed by the user through utilizing the OPC UA HistoryRead services in the OPC UA server implemented by the Historian or by reading directly from the database through SQL commands. In addition to collecting data values, the Historian creates entry points to all the added sources servers in its own address space. In this way, it acts as a simple aggregating server and enables the use of OPC UA services, such as Read, Write and Subscribe, in the aggregated address space.

3.2.2 Prosys OPC UA Java SDK

The Historian software utilizes the Prosys OPC UA Java SDK for its implementation of OPC UA functionality. Typically, an OPC UA application consists of three

software layers that are presented in the figure 12 below. The lowest level is a stack that enables the OPC UA communication between clients and servers. It implements the different transport mappings and functions for message serialization, security and transport. The stack is responsible for enabling the invocation of OPC UA Services. A stack can be implemented in any programming language and currently there exists implementations for ANSI C / C++, .NET and Java. These stacks are developed and maintained by the OPC Foundation. In addition, the are also open-source stacks implemented for Node.js and Python.

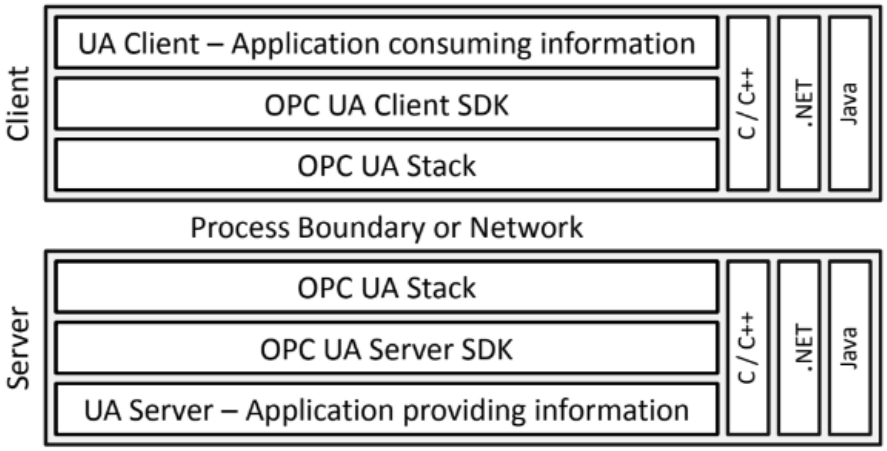

Figure 12: Software layers of a typical OPC UA application[29].

The second software layer in a typical OPC UA application is a software development kit (SDK). SDKs can also be implemented in various programming languages and currently there are solutions for ANSI C / C++, .NET, Java, Node.js and Python, similarly to the stacks. SDKs are usually commercialized by companies such as Unified Automation and Proys OPC. SDKs are software development tools that implement further abstractions and common functionality on top of the stacks. They aim to handle low-level functionalities and to hide excess complexity from the application programmer behind an easy-to-use application programming interface (API). SDKs can be utilized in the creation of server and client applications to reduce the development effort and speed up the development process.

The last software layer is the application layer. It implements the application-specific logic and functionality and is usually built using the API of one of the SDKs and using the same programming language as the SDK. The application layer defines, for example, if the application implements server or client functionality or both. It also dictates what services the application supports, how the address space of a server is constructed, what information models it implements and the functionality of methods, among other aspects.

The Prosys OPC UA Java SDK is a Java based SDK that utilizes the Java stack. It is divided into server and client SDKs that organize the components needed in the development of the respective OPC UA applications. It includes code generation capability for creating an address space from an imported information model as well

as many samples that can cater to common use cases. The Prosys OPC UA Java SDK is a commercial product with an openly purchasible licence.

3.2.3 Graphical User Interface & JavaFX

The Historian application contains a separate configurator part, called the Historian Configurator, that exposes a graphical user interface (GUI) through which the user can configure the parameters of the Historian application. The GUI also exposes some useful information for monitoring the status of the application and the configured source servers as well as the recorded Variables. The figure 13 shows a screen capture of the Historian Configurator with the tab for handling source servers open and having two configured servers.

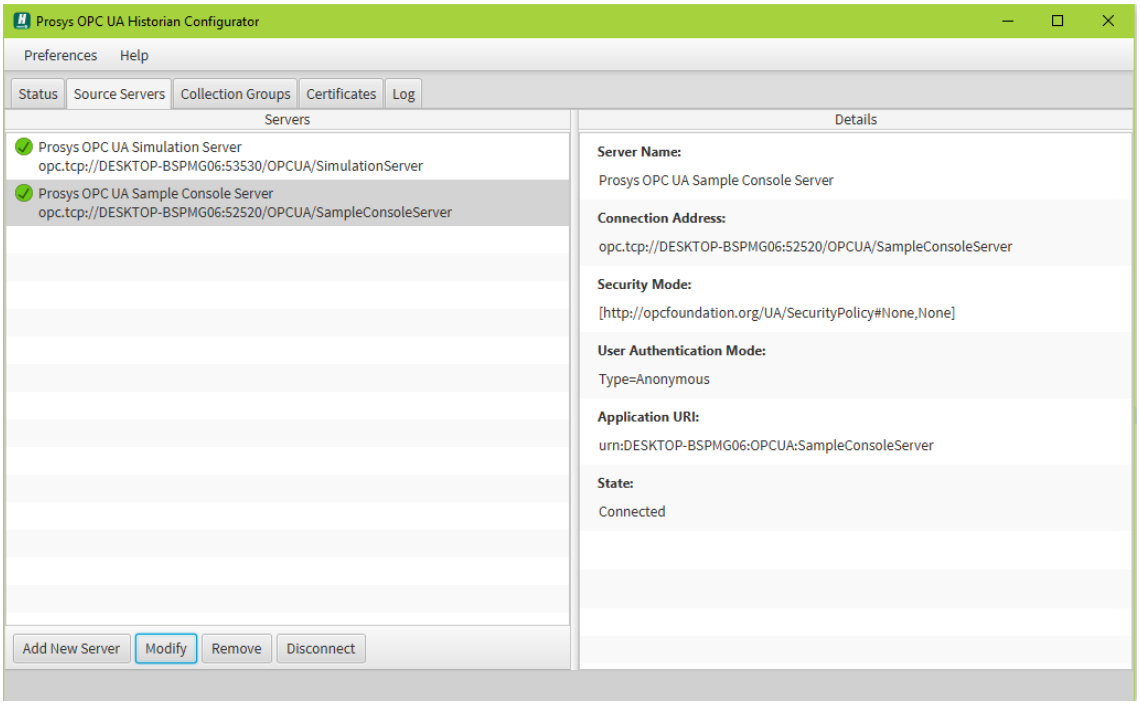

Figure 13: View of the Source Servers tab of the Historian Configurator.

The configurator is an independent application separated from the main Historian application. All commands given by the user in the configurator GUI are transmitted via OPC UA method services from the client in the configurator to the server in the main application. The configuration data is encoded as a JSON string that is used as the input argument for a method in the address space of the Historian server. The server receives this input string, interprets it and then performs the requested activity. Communication in the direction from the server to the client is conducted through a database connection that is further explained in the next subsection.

The GUI of the configurator is programmed using the JavaFX library. JavaFX is a Java-based API toolkit for building applications with diverse user interfaces. The JavaFX API is available as an integrated feature in the Java SE Runtime Environment (JRE) and the Java Development Kit (JDK) for Java 7 and 8. JavaFX

includes a set of GUI control elements that can be used to provide user input and a scene graph paradigm for creating the structure of the GUI components. The scene graph is a tree-like structure that maintains vector-based graphic nodes. JavaFX can interact with any Java code and therefore it can be connected as the front-end for existing Java applications. The connection between UI components and application functionality can be implemented by binding the UI to properties in a model or by using change listeners. JavaFX user interfaces can be developed by programming Java code or a special FXML language or by using a specialized Scene Builder software. FXML is an XML-based declarative markup language for constructing JavaFX-based user interfaces. The JavaFX Scene Builder is an application that allows the user to interactively design a graphical user interface that can then be generated as an FXML file.

3.2.4 SQL Database Connectivity

The Historian software stores all the collected historical values and its settings and configuration data to a Structured Query Language (SQL) database. SQL is a language standardized by ANSI, ISO and IEC for storing, manipulating and retrieving data in databases. SQL data is stored in catalogues that can contain several tables. Each table has a set of rows and columns that have set data types and store the actual data values. Data can be retrieved from or manipulated in the database by executing specified SQL statements with arguments that define the functionality.

The Historian supports Microsoft SQL Server, MySQL and MariaDB which are examples of relational database management systems (RDBMS). Hibernate ORM (object-relational mapping) framework is used for handling the mapping of Java classes to a relational database and persisting the data of Java objects. Hibernate is able to map Java data types to SQL data types and manage the database tables for storing the data. Hibernate API is database independent and it can internally handle the interoperability issues related to using different RDBMSs. Hibernate is free software that is distributed under the GNU Lesser General Public License.

3.2.5 Current Issues

Currently, the aggregating server functionality in the Prosys OPC UA Historian does not support any sort of user configurability. It automatically adds entry points to all the source servers that are added to the program. The entry points are added to a pre-defined location in the address space of the aggregating server and allow the user to access the 'Objects' folders of the source servers through these entry points. The aggregation keeps each source server strictly separated and is not able to understand any semantic connection between Nodes from different servers. Models from different servers that correspond to the same entities cannot be recognized.

Type consistency is an issue in the existing implementation of aggregation in the Prosys OPC UA Historian. When the Historian aggregates several source servers, it regards all type definitions as server-specific and creates new namespaces for them. This means that instances from different source servers that are originally of the

same type are transformed by the aggregation to the new server-specific types. After that, they are no longer semantically the same types. This leads to type consistency issues.

An example of a type inconsistency case is portrayed in the figure 14. It displays a client-side view of the aggregated address space of the Historian. Two different servers called 'Prosys OPC UA Sample Console Server' and 'Prosys OPC UA Simulation Server' are aggregated and they both contain identical Objects called 'MyDevice' (highlighted in the upper and lower parts of the figure). These Objects are instances of the same ObjectType named 'MyDeviceType' from a vendor-specific information model that is identified by the NamespaceUri 'http://www.prosysopc.com/OPCUA/SampleAddressSpace'. However, after the aggregation, each of the Objects have different type definitions which can be seen from the differing NamespaceIndices on the targets of the HasTypeDefinition References. These indices 20 and 26 point to new server-specific NamespaceUris created by the aggregation procedure for the types and instances located on each source server. After this procedure, the types are now semantically unequal. Furthermore, neither of the types no longer belongs to their original information model in the namespace 'http://www.prosysopc.com/OPCUA/SampleAddressSpace'.

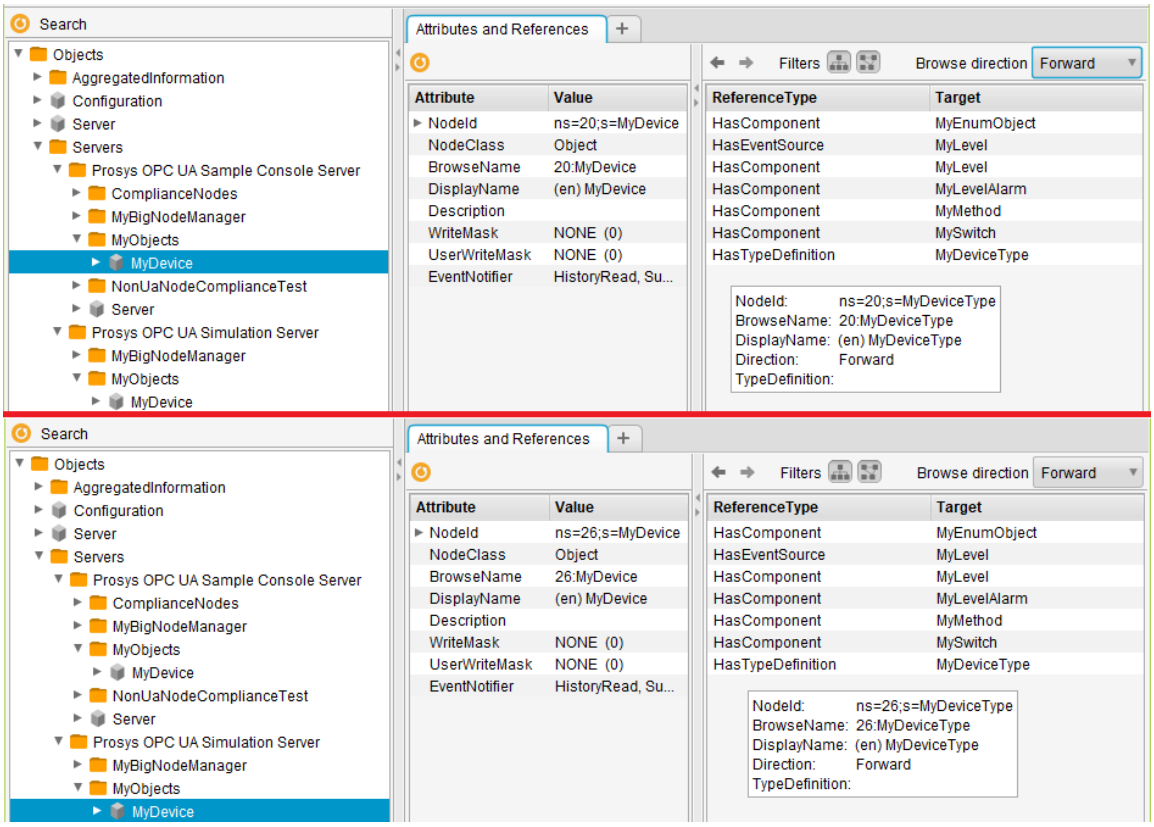

Figure 14: Example of a type inconsistency between two identical Objects.

By creating the new server-specific type definitions, the aggregation procedure disrupts the interoperability benefits brought by common information modelling standards. For example, a client might be configured to handle the instances of a

type from one of the companion specifications. After the aggregation procedure has changed all type definitions to the new server-specific definitions, the client will no longer recognize any instance as being an instance of the type it was configured to handle. Thus, it will not be able to handle the types accordingly.

3.3 Requirements

The goal assigned to this thesis was to design and implement a prototype aggregating OPC UA server that can integrate information from several sources into one OPC UA address space. The source servers may expose information from different application domains such as maintenance systems, design systems, control systems and product life cycle management systems. Therefore, the different models can implement various OPC UA information models and present aspects of the same entities. This needs to be taken into account in the aggregation. Because the goal is rather vague, it is necessary to define more detailed requirements before starting any development on an actual implementation. The following subsections will outline the different requirements defined for the new functionality. Industry representatives Karttunen and Rossi[37] from the company Outotec were interviewed to provide possible use cases and requirements. However, the described functionality is quite novel and ahead of the current development in the industry. Currently, many companies are only developing rather basic OPC UA implementations and do not see the described functionality as being immediately relevant. Due to these reasons, no relevant input was received from industry representatives. Therefore, the solution aims to create a generic solution for a wide array of use cases and the requirements are drafted solely based on the views of the writer of this thesis and of colleagues at Prosys OPC Ltd.

3.3.1 Programming Language and Software Platform

The software implementation developed during the course of this thesis should be built as a new feature in the Historian application. The aim is to improve the existing software and expand its functionality to make it more marketable and to meet a greater range of user needs. Additionally, a great amount of software development effort will be saved by utilizing the prefabricated constructs and functions of the existing Historian software. Therefore the new functionality should be developed using the same software platform as the existing implementation, namely Java programming language and the Prosys OPC UA Java SDK.

3.3.2 Performance & Usability

The developed software is meant to be only a prototype and performance is not a major concern. Nevertheless, the performance should be acceptable. In the context of this thesis, acceptable performance is defined as not compromising the usage of the software through excessive delays and processing.

In addition, the new functionality should be easy to use. According to general experiences across the employees at Prosys OPC, it has been realized that most users find the concepts and functionalities of OPC UA difficult to understand and

use. Therefore, users would appreciate a simple user interface and functions that are automated to a high degree. Performing as much functionality as possible without any user input should be a central goal to improve efficiency. To ease the deployment of the software in different application circumstances, it should not require any modifications to the source servers. Changing the address spaces of the source servers to accommodate aggregation would require a large amount of work and testing from the user along with possible downtime for the server. Consequently, it is not a viable option.

3.3.3 Type Consistency

The implementation developed in this thesis should fix the existing problems in the type definitions of the Historian and keep the types semantically consistent. In the scope of this thesis, semantic consistency of types means that when an instance is aggregated from a source server to the aggregating server, the characteristics of its type definition do not change. Before and after aggregation, the type of an instance should remain in the same namespace with the same NodeId value and possess similar characteristics, including the Attributes and Properties of the type. The type definition can also be a hierarchical structure with multiple References which should also remain unchanged.

For the implemented aggregation feature this also means that it should appoint the same type definitions to aggregated instances across all the source servers if the instances had the same type definition originally. The functionality should not create new namespaces or alter the original type definitions to server-specific ones so that all instances remain in their original namespaces and are associated to their original information model.

3.3.4 Aggregation Functionality & User Configurability

The created software solution must create an aggregated address space by accessing the defined source servers and applying an aggregation procedure on them. This aggregated address space should integrate information in a generic way as the solution should be applicable for general usage. It is impossible to know how or where the software will be used or how the information is modelled in those usage environments. Therefore, the aggregation procedure should be configurable by the user. The user should be able to define how the aggregated address space is constructed based on information that is modelled on the underlying source servers. This way the aggregation procedure is flexible and can meet the needs of various use cases. One definite use case requirement is the case where there are different models on different servers that express different aspects of the same entity. The aggregation procedure should be able to concentrate this information to offer an unified view of the entity. The aggregation algorithm must be able to function properly with data that implements any information model. This means that data conforming to any companion specification or even a vendor- or application-specific model should be aggregated properly.

4 Design & Implementation

This chapter will delve into the practical side of this thesis. The following sections will detail various aspects of the chosen approach for meeting the requirements defined in the previous chapter for the aggregating server feature in the Historian application. The first section will detail how the implemented aggregation procedure handles the aggregation of types. This is followed by a presentation of the architecture and the algorithms for handling the aggregation of instances. The final sections will detail how the aggregating server handles the mapping of services from the aggregated address space to the source servers and also present an overview of the architecture of the implementation.

4.1 Introduction

The information provided by an OPC UA server can be divided into types and instances. Type information consists of all the DataTypes, ObjectTypes, VariableTypes, EventTypes and ReferenceTypes in the address space of the source server. The type information defines the common characteristics for all the associated instances on the server. Instances model the actual data and functionality in the address space of the server in the form of Objects, Variables and Methods. The Address Space Model of the OPC UA specification defines the main structure of every OPC UA server. This structure is also presented in the figure 15. All of the type information is included under the folder named 'Types' while instance related data is concentrated under the 'Objects' folder. To implement the required aggregation functionality, this thesis took a three-part approach consisting of type aggregation, instance aggregation and new service mappings. These parts function together to form the new aggregating server feature in the Historian application.

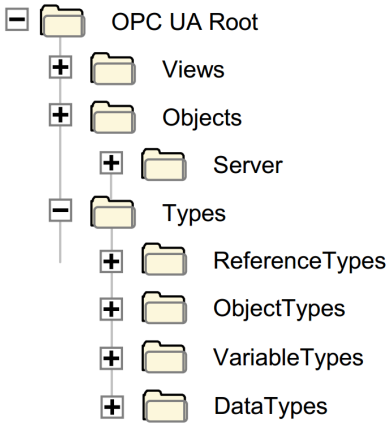

Figure 15: Standard structure of the address space of an OPC UA server[15].

4.2 Type Aggregation

To meet the type consistency requirements, the chosen approach was to aggregate the type definitions from all the source servers. This means that all new types that are discovered on the source servers are added to the address space of the aggregating server. As a result, all type definitions of the instances on the aggregating server can be made locally. Therefore, all equal type definitions from the various source servers will point to the same local types after aggregation. This will enable the types to remain consistent across the aggregated instances. Equality of types is evaluated based on the NamespaceUri and the NodeId value, which together form an unique identifier for a type definition Node. The overhead of adding the types to the server is negligible since the amount of different types is limited and the type aggregation is performed as a merge. This means that types already existing on the server will not be added.

The type aggregation procedure can access all the type information offered by a source server through the 'Types' folder. The hierarchy starting from this folder contains all the type definitions that the server utilizes. In addition, the Address Space Model defines a set of base types BaseDataType, BaseEventType, BaseObjectType, References and BaseVariableType that all other types inherit from. This type hierarchy and the base types are depicted in the figure 16. The known hierarchical structure of type inheritance is utilized by the type aggregation algorithm to merge the type definitions of added source servers into the type hierarchy of the aggregating server.

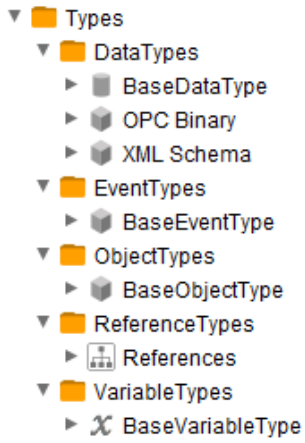

Figure 16: Top levels of the type hierarchy of an OPC UA server.

4.2.1 Type Aggregation Algorithm

The following is a stepwise description of how the implemented type aggregation algorithm proceeds when a new source server is added for aggregation:

1. Browse the BaseDataType, BaseEventType, BaseObjectType, References and BaseVariableType Nodes from the address space of the source server.

2. Browse the same Nodes from the address space of the aggregating server.

3. List all the HasSubtype References for each of the browsed Nodes from both servers.

4. Compare the HasSubtype References between the same Nodes on the aggregating server and the source server. The comparison is performed using the NodeIds of the target Nodes of the References. If the NamespaceUris and the values of the NodeIds are equal, then the target Nodes are regarded to be equal.

5. If no disparity between the References is detected, move to step 6. Otherwise, a new type has been found. The new type is the target of the HasSubtype Reference that was not found on the aggregating server. This new type is added to the aggregating server by creating a new Node and copying all the Attributes to the new Node and connecting it to the evaluated Node with a HasSubtype Reference.

   Because types can be complex hierarchical structures, it is necessary to evaluate all the References of the newly discovered type Node. If the Node has References other than HasTypeDefinition or HasModellingRule, it acts as a root Node for a hierarchical tree-like type definition structure. This type definition tree is copied in its entirety to the aggregated server by recursively following all forward References from hierarchy level to the next and copying the respective target Nodes. A type definition branch ends when there are no more References other than HasTypeDefinition or HasModellingRule. The HasTypeDefinition or HasModellingRule References point to other type definitions or to ModellingRule instances on the server. If these type definitions or ModellingRules already exist, the References should be copied to the aggregated server, but their target Nodes do not need to be added.

   However, it is also possible that the target of the HasTypeDefinition Reference is another new type which must be added to the server before copying the Reference. Similarly, the target Node of a HasModellingRule Reference can also be an instance that is not located on the aggregating server and should be added.

   Repeat step 5 for each new type definition discovered as a subtype of the currently evaluated Node.

6. Move down the type hierarchy by browsing all the target Nodes of the HasSubtype References. Repeat the algorithm starting from step 2 for all the newly browsed Nodes.

4.3 Instance Aggregation

The previous section presented how the aggregating server handles type definitions from its source servers. The aggregated information provided by multiple sources must be semantically coherent and therefore centralizing all type definitions to the

aggregating server is beneficial. This type information is needed in order to aggregate information from the source servers in the form of instances that are occurrences of these type definitions. The instances model the actual data on the source servers in the form of Objects and Variables and functionality in the form of Methods. To meet the requirements stated for the aggregation procedure, three key components were created. Firstly, aggregation rules provide constructs that enable the customization of the aggregation procedure. Secondly, a special information model was created for the use of the aggregated address space. Lastly, an algorithm was developed to perform the aggregation of information. The aggregation algorithm searches through the source servers and finds all instances that should be grouped together according to the aggregation rules. This allows for configuring the aggregation procedure to locate, for example, all models that describe the different aspects of the same entity. This was the requirement placed for the implementation to meet the information integration needs involving several heterogeneous providers of information. However, the aggregation rules make it possible to perform varied aggregation procedures also inside a single server and with similar information models.

4.3.1 Aggregation Rules

To meet the requirements for user configurability in the aggregation procedure, the concept of aggregation rules was developed. These rules define which instances from different servers are grouped together. By changing the rules that are used by the instance aggregation algorithm, it is possible to change the behaviour of the aggregation procedure. By allowing the user to change these rules, they can configure the aggregation procedure to meet different use-case-specific requirements, such as different information models or system parameters.

The following will outline the Java class architecture of the aggregation rules that was designed and implemented during the work in this thesis. The figure 17 below shows an UML diagram overview of the architecture. The model is greatly simplified in its presentation to highlight the central features.

AggregationRule models the concept of an aggregation rule. An AggregationRule contains a name attribute that can be used to identify the rule. It is useful to be able to clearly distinguish different rules because the amount of rules is not limited in any way. Additionally, the AggregationRule contains a number of *AggregationFootprints*. They are organized in lists based on the namespaces where they are applicable. This means that the AggregationFootprint is only evaluated against Nodes that reside in the specified namespace. However, it is also possible to define that the footprint is evaluated against all Nodes irrespective of their namespace. The applicable namespace is defined when a new AggregationFootprint is added to the rule using the addFootprint-method. The AggregationRule also contains methods for evaluating the rule against a given Node as well as a getter for the name attribute.

AggregationFootprint models all the properties that a Node must have so that it matches the footprint. These properties are described by *AggregationFeatures*. AggregationFootprint contains a list of all the AggregationFeatures that are associated with it and methods to set and get AggregationFeatures. It also contains a method

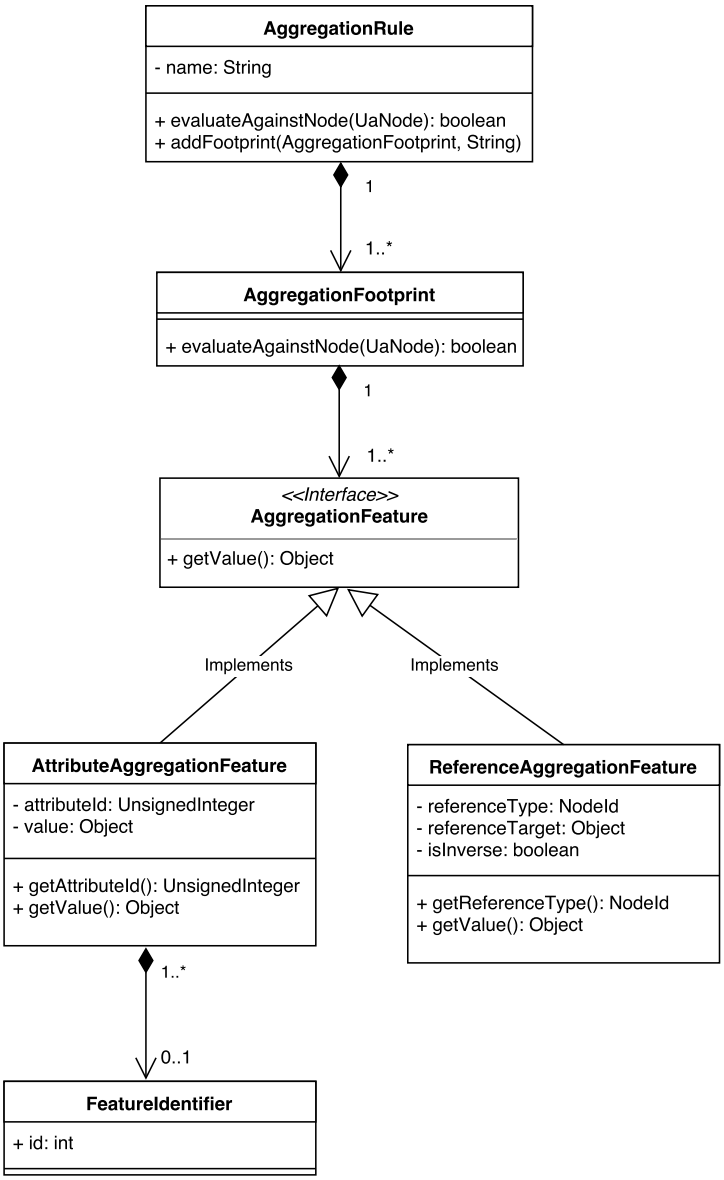

Figure 17: Java class architecture for aggregation rules.

that evaluates if a given Node matches the footprint.

AggregationFeature is a common interface implemented by *AttributeAggregationFeature* and *ReferenceAggregationFeature*. It defines one method for getting the value for a feature.

AttributeAggregationFeature contains an identifier that signals which Attribute of the evaluated Node the feature is concerned with. It also contains a variable with a value that signals what the value of the defined Attribute should be so that an evaluated Node matches the feature. Alternatively, instead of an explicit value, the variable can define an equality relationship between features. This is done using a special *FeatureIdentifier* class.

FeatureIdentifier is a simple class that contains an integer-valued ID for com-

paring its equality to other FeatureIdentifiers. When a FeatureIdentifier is assigned as the value of an AttributeAggregationFeature, it means that the value of the Attribute defined by the AttributeAggregationFeature is evaluated against the values of the Attributes defined in all other AttributeAggregationFeatures with the same FeatureIdentifier. This means that a FeatureIdentifier defines an equality condition between values of Attributes. Values of the Attributes are compared between Nodes and the Nodes with equal values for the Attributes marked by the same FeatureIdentifiers are matched together. In this way, FeatureIdentifiers make it possible to group instances based on common characteristics instead of just preset explicit values.

ReferenceAggregationFeature is similar to the AttributeAggregationFeature but it instead defines the ReferenceType and the target for a Reference that the evaluated Node should possess to match the feature. The defined value for the target of the Reference can be either a NodeId or another AggregationFootprint. The direction of the Reference is also defined. Through the use of ReferenceAggregationFeatures it is possible to build a complex hierarchical construct of multiple AggregationFootprints that can match the semantics of any OPC UA object.

The figure 18 below presents an example model that represents an AggregationRule implementing most of the features introduced above. The rule has a single AggregationFootprint called 'Footprint1' that applies to Nodes from all namespaces. The footprint contains one AttributeAggregationFeature called 'AttributeFeature1' that matches to all Nodes whose NodeClass is Variable. The footprint also contains one ReferenceAggregationFeature called 'ReferenceFeature1' that matches to all Nodes that have a Reference of type HasProperty. The reference feature has a Reference target that is another AggregationFootprint called 'Footprint2'. This means that for a Node to match the Reference feature, it must have a HasProperty Reference whose target Node matches the AggregationFootprint named 'Footprint2'. The 'Footprint2' footprint adds a requirement that the DisplayName of the evaluated Node must have the value 'Definition'. In addition, the footprint has an Attribute feature with a FeatureIdentifier called 'FeatureIdentifier1' for the Value Attribute. This means that only Nodes with equal values for the Value Attribute are grouped together.

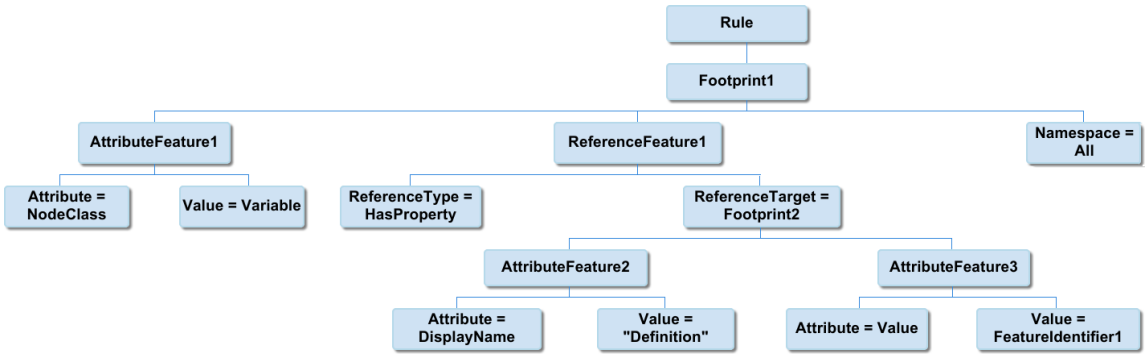

Figure 18: Example model of an AggregationRule that aggregates together all Variables that have a Property whose DisplayName equals 'Definition' and whose Values are identical.

4.3.2 Information Model for Aggregation

A special information model was developed to define how the information in the aggregated address space is constructed. This model is then implemented in the Historian application. The information model dictates how the address space of the aggregating server is structured. This structure is also presented in the figure 19. The model adds a folder named 'AggregatedInformation' under the 'Objects' folder defined in the OPC UA specification. This 'AggregatedInformation' folder acts as the entry point to the aggregated address space. Under it are four more folders named 'Equipment', 'PhysicalAssets', 'Material' and 'Personnel'. They categorize the aggregated information similarly to the ISA-95 Common Object Model. This concept was also discussed in the chapter 3. All the information modelled by the examined companion specifications can be mapped into these four categories. It is also a very general model that applies to various use cases, especially in the industrial environment. The implemented aggregation information model offers an example of a possible approach for organizing the aggregated address space and is not an entrenched implementation. It can be changed with minimal effort to meet different requirements.

The other Objects presented in the figure 19 are from the previous implementation of the Historian. The 'Configuration' Object and its 'Configure' Method are used by the Historian Configurator application to communicate new settings and parameters provided by the user to the Historian application. The 'Servers' Object contains folders that act as entry points to the address spaces of the source servers that were configured to the Historian application. The example shown in the figure 19 contains two source servers with user-provided names.

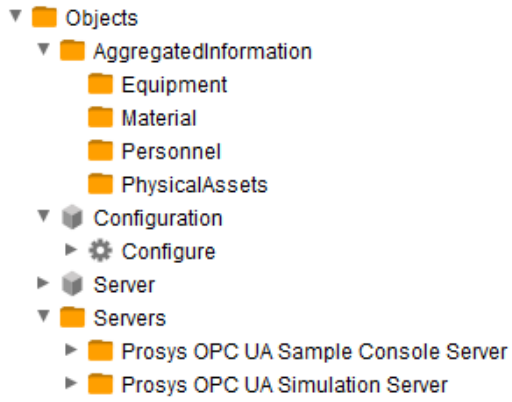

Figure 19: The aggregated address space of the Historian application.

In addition, the aggregation information model adds a new ObjectType called AggregateObjectType. This is a subtype of the BaseObjectType and is used to group different instances that model the same entity. The AggregateObjectType represents the entity and it has References to models from different source servers that model its various aspects. For this purpose the information model also defines new ReferenceTypes: HasModel, HasDIModel, HasADIModel, HasPLCopenModel,

HasAutomationMLModel and HasISA95Model. HasModel is a subtype of the Aggregates ReferenceType from the OPC UA specification and the other custom ReferenceTypes are subtypes of the HasModel ReferenceType. The new ReferenceTypes and their type hierarchy is displayed in the figure 20. The source Node of these ReferenceTypes is always an AggregateObject and the target Node is the root Node of a model that represents an aspect of the AggregateObject. The particular ReferenceType is chosen based on the namespace of the type definition of the target model. If it belongs to one of the companion specifications, then the ReferenceType is one of the specific types, for example, HasADIModel for the Analyser Devices information model. HasModel ReferenceType is used for the standard OPC UA namespace.

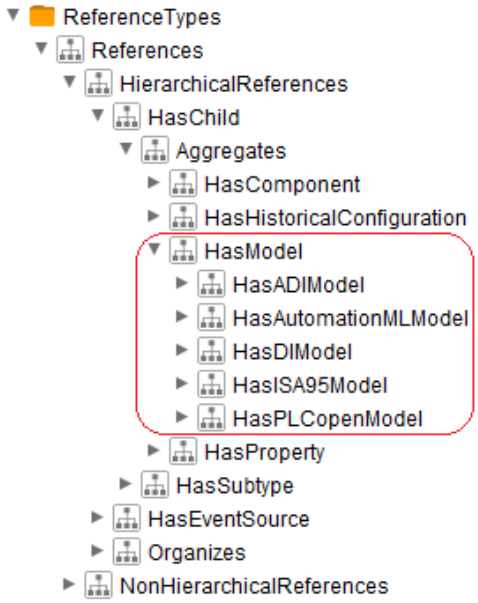

Figure 20: The new ReferenceTypes defined by the information model for aggregation.

4.3.3 Instance Aggregation Algorithm

The instance aggregation algorithm begins running when the type aggregation for the source server has been completed. The algorithm starts by browsing the standardized 'Objects' folder that organizes all the instances in an OPC UA server. The OPC UA specification defines that 'the intent of the 'Objects' Object is that all Objects and Variables that are not used for type definitions or other organizational purposes (e.g. organizing the Views) are accessible through hierarchical References starting from this Node'[15]. This concept is utilized by the instance aggregation algorithm to Browse through each Node in the instance hierarchy and evaluate if it is suitable for aggregation. After browsing the 'Objects' folder, the algorithm proceeds according to the description below:

1. List the target Nodes for all hierarchical References from the current Node.

2. Find out the NamespaceUri for each of the target Nodes from their NodeId. If a NodeId does not use a NamespaceUri, then the NamespaceUri must be deduced from the NamespaceIndex using the NamespaceArray of the source server.

3. Find all configured aggregation rules that apply to the namespaces of the target Nodes.

4. If there are no applicable rules, then jump to step 10. Otherwise continue to the next step.

5. Evaluate the applicable aggregation rules against the target Nodes.

6. If the rule evaluation passes for a Node, then the Node is added to a set containing all the Nodes that match the rule.

7. If the set already contains matching Nodes, then jump to the next step. If the Node is the first to match the rule, then a new Object of type AggregateObjectType is created in the aggregated address space. The type definition of the matching Node is browsed upwards in hierarchy by following inverse HasSubtype References until the type is part of a known information model. This will define the folder in the aggregated address space where the AggregateObject is placed according to the mappings presented earlier in this thesis. For example, instances of the AutomationML information model are placed under the 'Equipment' folder. With instances of the ISA-95 model, it is necessary to find out if the type is a subtype of one of the types specific to equipment, physical assets, material or personnel in order to define the appropriate folder. 'PhysicalAssets' folder is the default folder for Nodes that do not belong to any companion specification. The information model that the Node belongs to also dictates the ReferenceType used in the next step. The new AggregateObject will be given the same DisplayName as the first Node matching the rule.

8. Find the AggregateObject that is associated with the rule. Each AggregateObject is specific to an aggregation rule so that all Nodes matching a single rule are organized under the same AggregateObject. An exception to this is when the rule uses FeatureIdentifiers. In that case, multiple AggregateObjects can be created for a single rule and each of the AggregateObjects groups together only Nodes with common characteristics.

9. Add a Reference of type HasModel, or of a more specific type if the Node is an instance of a companion specification, from the corresponding AggregateObject to the matching Node.

10. Move down the instance hierarchy tree by browsing all the target Nodes of hierarchical References from the current Nodes and start over from step 1. This recursive algorithm ends for a certain branch of the instance hierarchy tree when no hierarchical References exist for a given Node or if the Node has already between evaluated against all applicable rules. The latter condition

is a safeguard against loops in the instance hierarchy. To avoid the algorithm endlessly proceeding through the loops, it is necessary to save the NodeIds of all evaluated Nodes and then prevent the algorithm from evaluating these Nodes again.

A special part of the instance aggregation algorithm is the evaluation of aggregation rules against Nodes in the address space of a source server. This procedure is performed inside the scope of the AggregationRule Java class. The algorithm has two inputs, the Node that is evaluated and its NamespaceUri. The Node is an instance of the UaNode Java class which is a complex construct used by the Prosys OPC UA Java SDK to model general OPC UA Nodes. UaNodes can be read from OPC UA servers by utilizing the functionality in the Prosys OPC UA Java Client SDK. An aggregation rule matches an instance Node in the source server if any of its footprints matches the Node. Therefore, the evaluation of the rule starts by finding all the footprints that are applicable for the NamespaceUri of the evaluated Node. Then each applicable footprint is evaluated according to the algorithm that is detailed step-by-step below:

1. Access all the defined AttributeAggregationFeatures in the AggregationFootprint and read the values of the Attributes specified by the features from the source server.

2. If the value specified by an AttributeAggregationFeature is not a FeatureIdentifier, then compare the value of the Attribute that was read from the source server to the value specified by the AttributeAggregationFeature. If the values are equal, then the evaluated Node matches the feature. If the values are not equal, the Node is not a match and the algorithm ends for the current footprint. If the value specified in the AttributeAggregationFeature is a FeatureIdentifier, then the read value should be saved to a container. This container saves the values of FeatureIdentifiers specific to a certain Node for later comparison between different Nodes.

3. Access all the ReferenceAggregationFeatures defined for the footprint. If the rule does not have any, then the evaluated Node is a match for the footprint. If the AggregationFootprint has ReferenceAggregationFeatures, then it is hierarchical and the evaluated Node should be searched for References whose type matches the ones specified by the ReferenceAggregationFeatures. If the required References are not found, then the Node does not match the footprint and the algorithm ends. If a matching Reference is found, then continue to the next step.

4. Browse the target Nodes for the References specified by the ReferenceAggregationFeatures and start the algorithm from step 1 to evaluate each of the AggregationFootprints specified by the ReferenceAggregationFeatures against the target Nodes of the specified References. In this recursive way, the entire hierarchical AggregationFootprint structure should be evaluated against a Node

hierarchy on the source server. If any of the features in the hierarchy do no match, then the algorithm ends.

Alternatively, a ReferenceAggregationFeature might define a NodeId that should be compared against the NodeId of the target Node of the specified ReferenceType. If the NodeIds are equal, then the feature matches the evaluated Node, otherwise the algorithm ends.

5. The algorithm is completed and the Node is a match to the AggregationFootprint.

If the AggregationRule uses any FeatureIdentifiers, the values of the Attributes marked by the FeatureIdentifiers must be compared against all the other Nodes matching the rule. Only if the values marked by the same FeatureIdentifiers are equal between Nodes, are they grouped together in the aggregated address space.

4.4 Service Mappings

The last component of the three-part solution implemented in this thesis is the mapping of OPC UA Services. This relates to how the aggregating server handles service calls, such as Browse, Read, Write and Subscription, made in its address space. As was presented in the overview of aggregating OPC UA servers in the second chapter, the aggregating server functions by transmitting the service calls it receives to the underlying source servers. Much of this basic architecture is already implemented in the existing version of the Historian software. The Historian employs a typical service mapping mechanism and an aggregating server architecture similar to the one presented in the second chapter. The details of the implementation are presented in the master's thesis by Asikainen[28] on which the actual software is based on, however, with some modifications and further development. The existing software already enables using Browse, Read, Write, Subscription and Method services in the aggregated address space.

Nevertheless, some modifications needed to be made to the implementation of Browse services to facilitate the use of the type definitions on the aggregating server together with the aggregated instances. When a Browse request is made to the aggregating server, it is transformed and relayed through one of the internal clients to a server that is the source for the aggregated Node. Transformation of service requests is necessary because information is modelled differently between the aggregated server and the source server. Similarly, the response received from the source server by the internal client must be transformed to a form that is suitable for the aggregating server. However, in the existing implementation of the Historian software all Browse results were transformed to a form where the types are server specific. This led to the type consistency issues presented earlier in this thesis. In addition, all ModellingRules were mapped to server-specific ModellingRules instead of common ModellingRules.

The aggregation feature developed during this thesis fixes the existing problems by adding new service mappings for Browse requests. The feature is plugged into the

existing Historian to handle all Browse requests regarding types or ModellingRules when they are performed in the aggregated address space. These particular Browse requests are identified by their service context that includes the ReferenceTypes HasTypeDefinition or HasModellingRule. The new Browse service implementation will transform all the NodeIds and BrowseNames of the respective service results to match types that are local to the aggregating server but correspond to the original types on the source server. Similarly, ModellingRules in service responses are mapped to local ModellingRules on the aggregating server. The correspondence between types on the source server and types on the aggregating server is evaluated on the basis of matching NamespaceUris and NodeId values. The NodeIds and BrowseNames of instances are not changed because instances must be server specific in order to distinguish similar Nodes between different servers.

The service mappings combine together the type aggregation procedure that guarantees that all needed type definitions are present locally on the aggregating server and the instance aggregation procedure that requires the use of consistent type definitions between instances. Cooperating together, these three components form the implemented solution.

4.5 Architecture

The new aggregation related functionality was created as a modular structure that was added to the existing Historian software. This means that most of the developed code is concentrated to a single Java package. In addition, the functionalities of the type aggregation, instance aggregation and service mappings are concentrated into a single central Java class named AggregationManager. The AggregationManager acts as the interface to the functionality and can handle the aggregation related functionality fairly independently. This aggregation management module can be plugged into the existing application with minimal code changes. Due to this modular structure, managing the code is simple. Changing implementation specifics related to the aggregation feature is easy because of the centralized code base. Similarly, tracking of errors and debugging is simplified. In addition, modularity makes it easy to enable or disable the new aggregation feature.

In the developed implementation, the Historian application includes a single instance of the AggregationManager that is able to handle the aggregation of all the added source servers. Aggregation can be enabled or disabled for each source server in the Historian separately. Some additions were made to the GUI of the Historian Configurator application to allow setting this option through the user interface. The new source server configuration view of the Historian Configurator is displayed in the figure 21. Changes were also made to the SQL database used by the Historian so that the aggregation setting can be persisted in the database and is remembered between application runs. When aggregation is enabled for a source server, then during startup the Historian will pass the client connected to the source server to the AggregationManager. The AggregationManager will then perform the type and instance aggregation procedures utilizing OPC UA Services through the connected client. The AggregationManager also handles all user-defined aggregation rules and

the creation of the aggregated address space. In addition, it is plugged in to the Browse services of the OPC UA server in the Historian application so that it can handle all the needed service mappings that were described in the previous section.

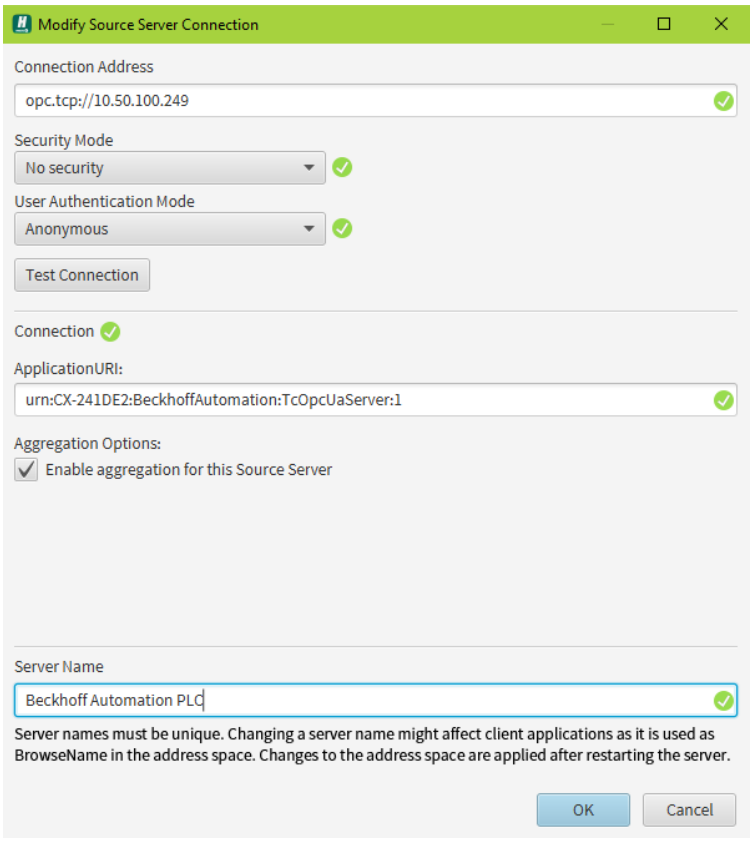

Figure 21: Source server configuration view of the Historian Configurator application with the setting for enabling or disabling the aggregation.

5 Evaluation

The following chapter will evaluate the implemented software feature by analysing its achievements and shortcomings and will present some possible approaches for future development. The first section presents test cases where the software is tested with aggregation of real OPC UA servers utilizing different information models. This section is followed by a general overview of the solution and analyses of the type and instance aggregation procedures.

5.1 Test Cases

5.1.1 Test Case for Type Aggregation

The first case used to evaluate the implemented aggregation feature tests the functionality of the type aggregation algorithm. For the test, two source servers were used. The first one was an integrated OPC UA server of a Beckhoff Automation PLC device and the second one was a development prototype version of the Prosys OPC UA Simulation Server software. Both of the servers implemented some of the OPC UA companion specifications in addition to their application- or vendor-specific information models. The table 1 below displays the information models implemented by the source servers. The version of the Simulation Server used in the tests supports loading NodeSet–files into the address space of the server. NodeSet is an XML Schema that characterizes a standard syntax that can be used to formally define information models and enables exchanging the model between computer programs[38]. The listed companion specifications were loaded into the Simulation Server from NodeSet–files provided by the organizations governing the specifications. The Beckhoff Automation PLC server was factory-configured to include the listed companion specifications.

Table 1: The standard information models, including the OPC UA base Information Model and the companion specifications, implemented by the tested source servers. The abbreviation 'SS' stands for the Prosys OPC UA Simulation Server and 'BA' for the Beckhoff Automation PLC.

Standard information models implemented by the tested source servers			
Specification	NamespaceUri	SS	BA
OPC UA	http://opcfoundation.org/UA/	✔	✔
ADI	http://opcfoundation.org/UA/ADI/	✔	
AutomationML	http://opcfoundation.org/UA/AML/	✔	
DI	http://opcfoundation.org/UA/DI/	✔	✔
PLCopen	http://PLCopen.org/OpcUa/IEC61131-3/		✔
ISA-95	http://www.OPCFoundation.org/UA/2013/01/ISA95	✔	

The test case has two central characteristics. Firstly, the types are aggregated

from more than one source server. The aggregation is performed simultaneously on both servers in different application threads, which poses greater demands for the aggregation algorithm. Secondly, the type definitions provided by the servers are dissimilar because the source servers implement different information models as seen in the table 1. On the other hand, there are also overlappings in the information models and therefore in the type definitions provided by the servers. Overlapping type definitions are also an important aspect to test. All OPC UA servers implement the standard OPC UA Information Model, which in itself creates overlapping types between servers. Additionally, both of the source servers in the test scenario also implement the Device Integration information model.

The tested source servers were run on different platforms. The Simulation Server was executed in the same computer as the Historian application while the Beckhoff Automation server was executed on the PLC device that wass located in the same local area network (LAN) as the other applications. The Beckhoff Automation server communicating over the LAN using Ethernet has higher network delays compared to the Simulation Server. In addition, the two source servers have varying hardware and calculation power.

The testing process was setup by adding the two source servers to the Historian application with the setting for aggregation being enabled. Then the Historian application was restarted five times to get five clean runs of the aggregation process. Some code was added to the aggregation algorithm to report the time taken taken by the type aggregation procedures for each of the servers. The measured times are reported in the table 2 below.

Table 2: The measured results for the time taken by the type aggregation procedure of the two source servers during five different test runs.

Time taken by the type aggregation procedure (in seconds)		
Aggregation run	Simulation Server	Beckhoff Automation PLC
1st	11.119 s	18.525 s
2nd	8.217 s	18.426 s
3rd	7.817 s	18.193 s
4th	7.862 s	18.221 s
5th	9.131 s	18.451 s

The namespaces implemented by the Historian were also recorded before and after the aggregation by reading the values in the NamespaceArray of the OPC UA server. The results are presented in the table 3 along with the application- and server-specific namespaces implemented by the two source servers. They present the namespaces used by the source servers in addition to the companion specifications listed earlier in table 1. The list of namespaces of the Historian after the aggregation shows that all namespaces containing type definitions were added from the source servers to the Historian unchanged. These namespaces include the companion specifications but also vendor- and application-specific namespaces if they contain type definitions.

Table 3: The effect of type aggregation on the namespaces of the Historian.

Namespaces of the Prosys OPC UA Historian before the type aggregation
http://opcfoundation.org/UA/
urn:DESKTOP-BSPMG06:ProsysOPC:Historian
http://www.prosysopc.com/OPCUA/HistorianAddressSpace
http://www.prosysopc.com/OPCUA/Gateway
Application-specific namespaces of the Beckhoff Automation PLC
urn:CX-241DE2:BeckhoffAutomation:TcOpcUaServer:1
urn:CX-241DE2:BeckhoffAutomation:Ua:PLC1
http://Beckhoff.com/TwinCAT/TF6100/Server/Configuration
Application-specific namespaces of the Prosys OPC UA Simulation Server
urn:DESKTOP-BSPMG06:OPCUA:SimulationServer
http://www.prosysopc.com/OPCUA/SampleAddressSpace
http://www.prosysopc.com/OPCUA/ComplianceNodes
http://www.prosysopc.com/OPCUA/ComplianceNonUaNodes
http://www.prosysopc.com/OPCUA/SimulationNodes
http://www.prosysopc.com/OPCUA/SampleBigAddressSpace
http://www.prosysopc.com/OPCUA/SimulationConfiguration
Namespaces of the Prosys OPC UA Historian after the type aggregation
http://opcfoundation.org/UA/
urn:DESKTOP-BSPMG06:ProsysOPC:Historian
http://www.prosysopc.com/OPCUA/HistorianAddressSpace
http://www.prosysopc.com/OPCUA/Gateway
http://opcfoundation.org/UA/ADI/
http://PLCopen.org/OpcUa/IEC61131-3/
http://opcfoundation.org/UA/DI/
http://www.OPCFoundation.org/UA/2013/01/ISA95
http://www.prosysopc.com/OPCUA/SampleAddressSpace
http://opcfoundation.org/UA/AML/
http://www.prosysopc.com/OPCUA/SimulationConfiguration
http://www.prosysopc.com/OPCUA/SampleBigAddressSpace
urn:CX-241DE2:BeckhoffAutomation:TcOpcUaServer:1

An example of the resulting type definition hierarchy in the aggregated address space of the Historian server is presented in the figure 22. It shows, on the left, the subtypes of the 'BaseObjectType' present before the aggregation procedure and then, on the right, the expanded type hierarchy after the aggregation with several new types added from the source servers. The new types on the right side of the figure display examples from all the companion specifications. 'AccessoryS-lotType' is from the ADI information model, 'CAEXBasicObjectType' is from the AutomationML model, 'TopologyElementType' is from the DI model, 'SFCType' is from the PLCopen model and 'ISA95ClassType' is from the ISA-95 model. A

noteworthy aspect of the figure is also that the highlighted 'AccessorySlotType' is correctly located in its standard-defined namespace marked by the NamespaceUri 'http://opcfoundation.org/UA/ADI/'.

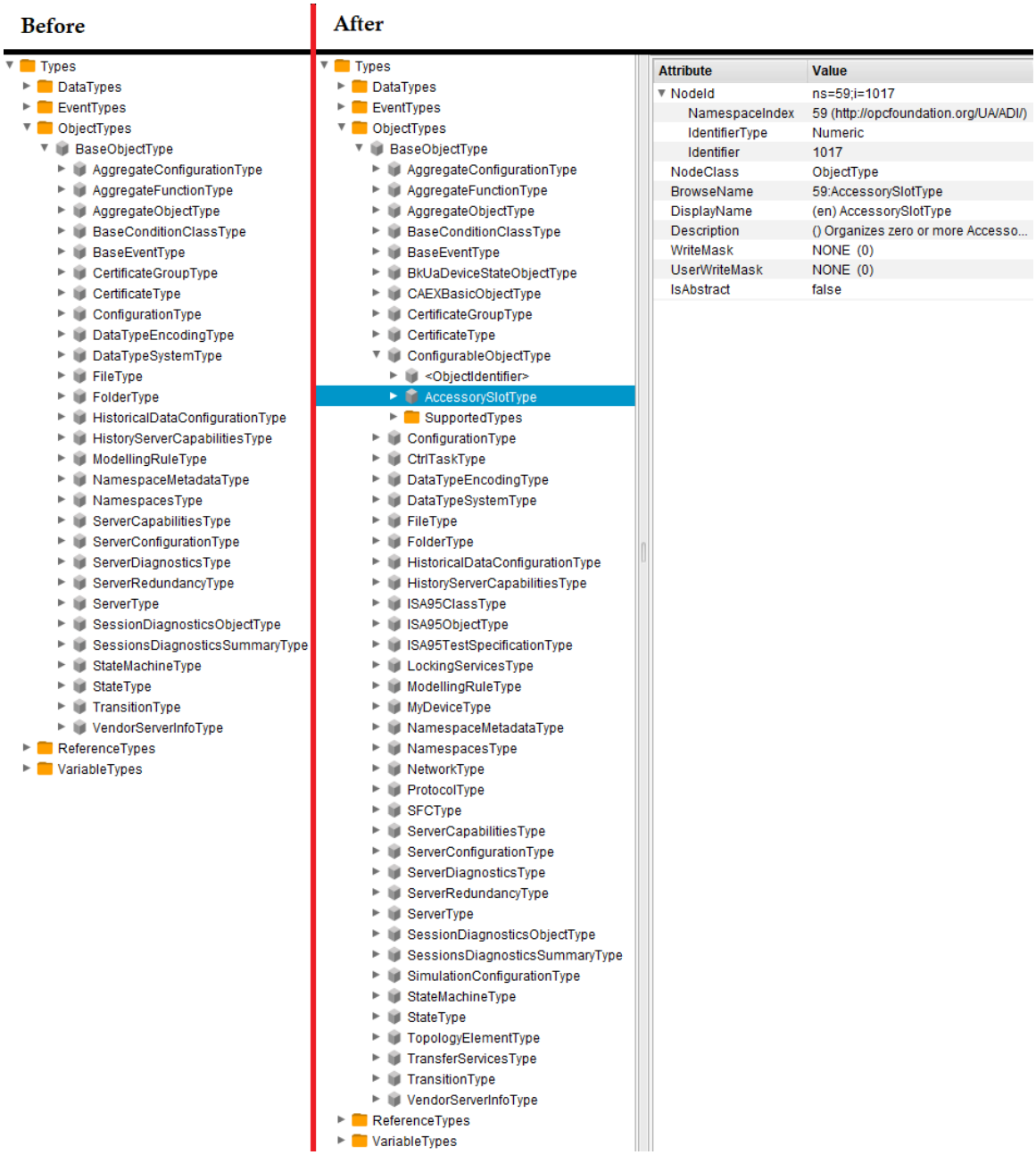

Figure 22: The ObjectTypes branch of type definition hierarchy tree of the Historian before and after the type aggregation. Also showing highlighted the 'AccessorySlotType' from the Analyser Devices companion specification as an example of a type that was successfully aggregated.

In the test, error-checking is already performed when the aggregation algorithm adds the aggregated type Nodes to the address space of the Historian. The aggregation

is performed using the Prosys OPC UA Java SDK that implements built-in sanity checks that do not allow the user to create any constructs that conflict with the OPC UA specification and will notify with error reports if illegal operations are attempted. During testing, the aggregation procedure was observed to run without any errors. In addition, a number of manual tests were performed by browsing various type definitions from the address space of the Historian and comparing their Attributes, Properties and References to the definitions provided by the companion specification documentations. No discrepancies were detected.

Finally, a special algorithm was developed that is able to go through the type definition hierarchies of two servers and report any differences between similar Nodes in the hierarchy tree of the two servers. The algorithm regards Nodes similar if they have equal NodeIds, i.e. same NamespaceUri and NodeId value, and then proceeds to compare their Attributes and References. The algorithm was used to compare the Historian server after the described aggregation procedure with the Simulation Server after all the companion specifications were loaded into its address space from NodeSet–files. In this case, the type definition hierarchies of the two servers should be the same. Nevertheless, some differences were reported by the algorithm. After manual inspection, it was revealed that these inconsistencies were caused by slightly different versions of the DI information model between the two source servers of the Historian. Because the aggregation procedure is additive, when two similar type definition Nodes with different References are aggregated, it results in the new type definition to contain References that are the sum of the References of the two source type definitions. This is the form of the inconsistencies that were noticed by the checking algorithm, however the amount of errors was small. In addition, some differences were noticed in the Attributes of type hierarchy Nodes between the cheched servers. The aggregation procedure prioritizes the Attributes of the first encountered type definition Node. If another server provides a type definition Node with an equal identity (NamespaceUri and NodeId value) but different Attributes, the deviating Attributes are ignored by the aggregation procedure. These problems were again caused by the different versions of the same information model between different source servers. For example, some types were abstract in one version but not in the other version.

The elapsed times measured for the type aggregation procedure show insight into the efficiency of the aggregation procedure. Nevertheless, the duration is dependent on the size of the type hierarchy tree of the source server that is being aggregated. Because the aggregation is run in parallel for all the configured source servers, the total time of aggregation is dictated by the slowest time for a single server. In all the test cases, the slowest aggregation time was observed for the Beckhoff Automation PLC server. The measured time in all the five test runs was consistently between 18.2 to 18.5 seconds, while the type aggregation for the Simulation Server fluctuated between 7.8 and 11.1 seconds. The amount of Nodes in the type definition hierarchy of the Simulation Server is higher than that of the Beckhoff Automation PLC. This would suggest a longer time needed for the type aggregation. However, the Simulation Server was faster in all cases. This is probably due to the Simulation Server being run on the same computer as the Historian. The Beckhoff Automation

PLC is accessed over the local network and, as a consequence, the communication is probably hampered by network delays in transmitting the OPC UA service calls and possibly by also slower hardware in processing the calls. Overall, the performance of the type aggregation was quite reasonable, especially since the procedure must be performed only once during the runtime of the application.

5.1.2 Test Case for Instance Aggregation

The instance aggregation test case is divided into two parts. The first part tests cross-server aggregation with two source servers but similar address spaces. Conversely, the second part tests cross-model aggregation with one source server but dissimilar address spaces implementing different information models.

The first instance aggregation test features two source servers: the Prosys OPC UA Simulation Server and the Prosys OPC UA Sample Console Server. They both implement similar address spaces with some sample instances but no added companion specifications. The implemented namespaces are listed under the 'Application-specific namespaces of the Prosys OPC UA Simulation Server' in the table 3. A simple aggregation rule was used in the test case. It applies to the namespace 'http://www.prosysopc.com/OPCUA/SampleAddressSpace' and aggregates together all Nodes that have a type definition with the DisplayName 'MyDeviceType'. A model of this rule is presented in the figure 23. It was known beforehand that there is one such instance on each of the servers. Therefore, the expected result for the test was that these two instances would be aggregated together and located in the aggregated address space after the procedure is completed.

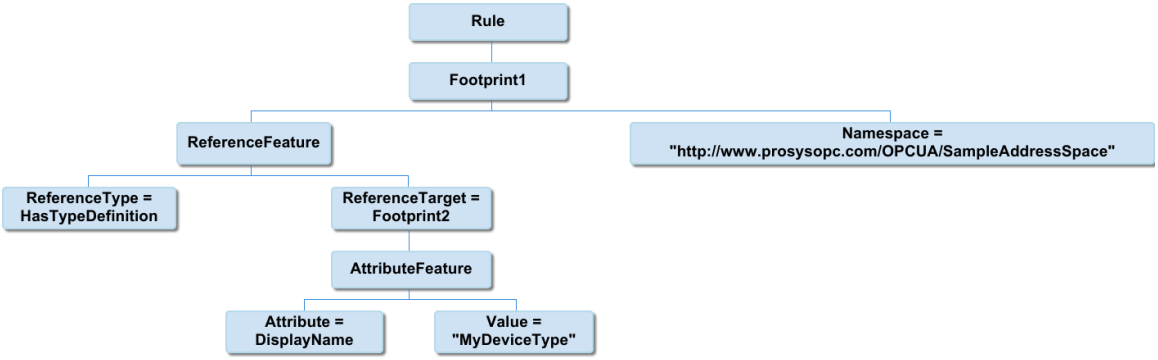

Figure 23: Model of the aggregation rule used in the first part of the instance aggregation test case.

Table 4 first presents the new namespaces added to the NamespaceArray of the Historian by the aggregation mechanism for the use of the instance aggregation. An instance from one server can have the same NodeId as an instance from another server. This is also true in the test case where the address spaces of the two source servers are mainly copies of each other. Nevertheless, the aggregating server must be able to recognize such instances from each other. Therefore, the Historian creates new server-specific namespaces for the instances. The NamespaceUri of an aggregated instance is formulated as the sum of the applicationUri of the source server

and the NamespaceUri of the Node on the source server. The applicationUri is defined by the OPC UA specification as a globally unique identifier for an application instance and can therefore be used to reliably identify different servers[14]. For the servers in the test case, the applicationUri is of the form 'urn:HN:SimulationServer' and 'urn:HN:SampleConsoleServer' where the 'HN' stands for the HostName of the computer running the servers. In the table 4, additionally the respective ApplicationNames 'SimulationServer' and 'SampleConsoleServer' are abbreviated to the forms 'SS' and 'SCS' to save space.

Table 4: The additions made by the instance aggregation to the namespaces of the Historian in the first part of the test case.

Instance-specific namespaces used by the Historian after the instance aggregation
urn:HN:SS/http://opcfoundation.org/UA/
urn:HN:SS/urn:HN:SS
urn:HN:SS/http://www.prosysopc.com/OPCUA/SampleAddressSpace
urn:HN:SS/http://www.prosysopc.com/OPCUA/ComplianceNodes
urn:HN:SS/http://www.prosysopc.com/OPCUA/ComplianceNonUaNodes
urn:HN:SS/http://www.prosysopc.com/OPCUA/SampleBigAddressSpace
urn:HN:SCS/http://opcfoundation.org/UA/
urn:HN:SCS/urn:HN:SCS
urn:HN:SCS/http://www.prosysopc.com/OPCUA/SampleAddressSpace
urn:HN:SCS/http://www.prosysopc.com/OPCUA/ComplianceNodes
urn:HN:SCS/http://www.prosysopc.com/OPCUA/ComplianceNonUaNodes
urn:HN:SCS/http://www.prosysopc.com/OPCUA/SampleBigAddressSpace

A view of the aggregated address space of the Historian is displayed in the figure 24. The upper part of the figure shows that the two instances called 'MyDevice' from different servers were successfully aggregated under an instance of the AggregateObjectType with the same name. The two aggregated instances are connected to the AggregateObject with the more general ReferenceType HasModel, because their type definition does not belong to or extend any companion specification information model. The lower part of the figure shows how these aggregated instances can be used as entry points to access information from the source servers, such as reading the value of the 'MyLevel' Variable that is highlighted in the figure. The lower part of the figure also shows that the type definition of the 'MyLevel' Variable points to a type in the namespace 'http://www.prosysopc.com/OPCUA/SampleAddressSpace' that is local to the aggregating server and was generated by the type aggregation algorithm.

The second part of the instance aggregation test case concentrates on the aggregation of multiple heterogeneous information sources that model different aspects of the same entity. The test is performed by first using the UaModeler software from Unified Automation to create two models that implement different companion speci-

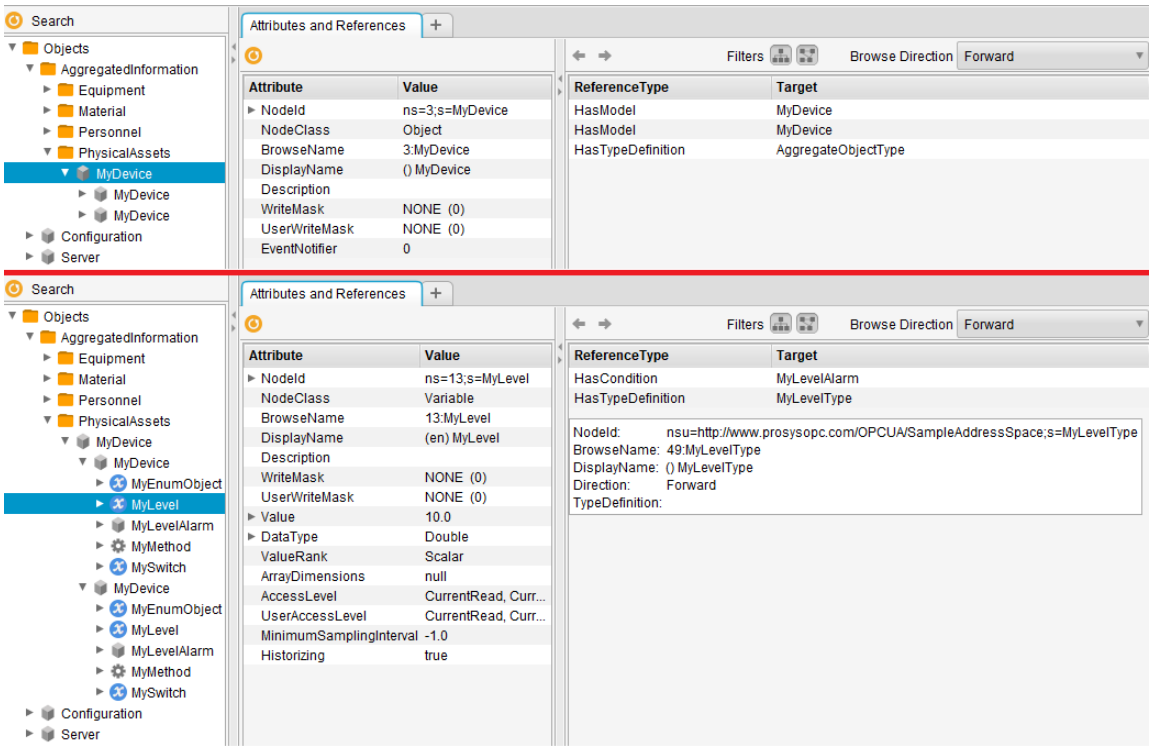

Figure 24: View of the aggregated address space for instances in the Historian after the instance aggregation is completed in the first part of the test case.

fications but model the same entity. These models are then saved as NodeSet–files and loaded to the Prosys OPC UA Simulation Server that is used as the source server for the aggregation procedure. One of the models utilizes the AutomationML and the other utilizes the Device Integration companion specification to create mock-up representations of a very simple manufacturing system from two different viewpoints. Both models create an instance called 'ManufacturingSystem' that contains two elements named 'firstScrewdriver' and 'secondScrewdriver'. However, each of the models displays different aspects of these elements and therefore use different type definitions that extend from their respective companion specifications. The AutomationML model exposes role-based characteristics of the elements, while the model implementing the DI companion specification displays parameters, such as the manufacturer and the serial number, of the actual physical device fulfilling that role. The testing then proceeds with the definition of an aggregation rule in the Historian that aggregates these two models together based on their namespaces and DisplayNames. The figure 25 shows that the aggregation was successfully completed and the instances are under the AggregateObject called 'ManufacturingSystem' with their correct information model–specific ReferenceTypes 'HasDIModel' and 'HasAutomationML'. The AggregateObject 'ManufacturingSystem' is placed both under the 'Equipment' and the 'PhysicalAssets' folders according to the information model mappings, because the Object groups models that belong to each of the categories.

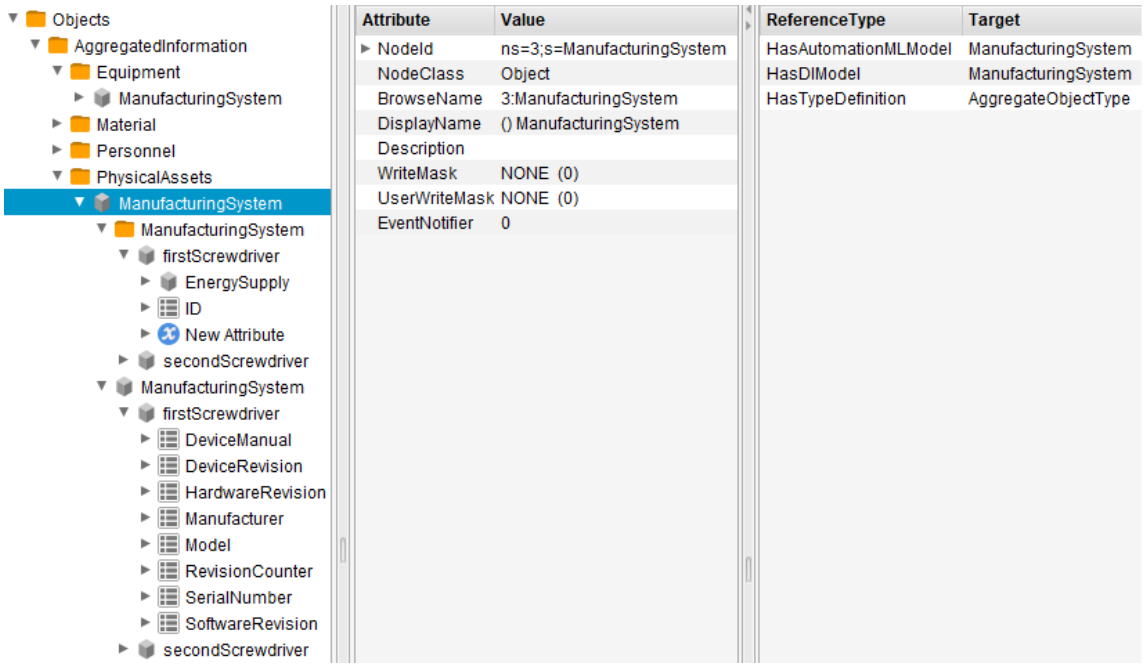

Figure 25: View of the aggregated address space for instances in the Historian after the instance aggregation is completed in the second part of the test case.

5.2 Solution Overview

The aggregation feature developed during the course of this thesis manages to create a proof-of-concept that meets the desired functionalities. The feature was added as a modular extension to the existing Historian software and successfully fixes the type consistency issues that were troubling the aggregating server of the Historian. In addition, the new aggregation feature manages to provide a fairly simplistic and generic solution to the problem of aggregating various heterogeneous models from different source servers. An important aspect is that the aggregation procedure utilizes aggregation rules that are user configurable and enable the solution to meet demands of varied use cases. Therefore, the main requirements of type consistency, user configurability and information integration are well met.

The solution is also highly automated and does not require user input after the aggregation rules have been configured and the aggregation is enabled. This improves usability and efficiency. The aggregation functionality also does not require any changes to made to the source servers. The aggregation procedure is entirely managed by the aggregating server, which makes it much easier to introduce the aggregation functionality into existing systems.

5.3 Evaluation of Type Aggregation

The type aggregation works as intended in keeping all types local to the aggregating server and assigning these type definitions to the aggregated instances so that they have semantically identical types. However, the aggregation of types has

some complications that are not properly addressed. Different source servers might implement different versions of the information models or companion specifications. These different versions may change the characteristics of the types and therefore information models may not be compatible between versions. Ideally, information models would be designed to be always compatible between earlier versions, but in reality this is not always possible or a realistic assumption. Currently, the aggregation procedure performs an additive merge of the type hierarchies between the aggregating server and the newly added source server. Nonetheless, if the new source server does not merely expand the existing type hierarchy but also changes the upper levels, then it will result in a type conflict. Types are compared based on their NamespaceUri and NodeId value, which can be read directly from the NodeId Attribute of a Node or deduced based on the NamespaceIndex. A type conflict means that type Nodes with equal identifiers between the aggregating server and the source server have some mismatch in their characteristics, such as Attributes or References. In the implemented version, the types existing on the aggregating server have priority over added types. Therefore, new source servers cannot alter the existing type hierarchy but only expand it. The aggregation procedure will not allow adding multiple types with equal identifiers. This can also lead to the changes in the type definitions of instances as the type definition from a source server is replaced by a definition on the aggregating server with altered characteristics. It is also possible that a new source server changes the characteristics of an existing type on the aggregating server through new References, such as by adding new Properties. The described cases can lead to the alteration of type characteristics after aggregation and to issues in using the aggregated type information. In future development, it could be feasible to request user input when a type conflict is detected or allow the user to change the parameters of the aggregation procedure and, for example, choose between additive or replacing merge options. Default parameters for the application cannot be set so that they would be suitable for every use case and therefore some user modification should be possible.

Further improvements to the type aggregation could be added in later development. The type aggregation could be optimized in various ways. However, the size of the type hierarchy is usually not very large and the aggregation process was found to be quite quick. But it would be possible to read the NamespaceArray of source servers to find out what information models they use. If they contain any known companion specifications, the type definitions could be loaded directly from a NodeSet–file without browsing the type hierarchy from the server. However, vendor- and application-specific types would still have to be copied from the source servers. Another possible option is to only add the type definition hierarchies of instances that are aggregated during the instance aggregation procedure. The current implementation adds a large amount of type definitions that might not be utilized in the context of the aggregating server. By optimizing the procedure to only aggregate types needed by the instance aggregation, it would be possible to greatly reduce the copying of unnecessary types. However, the overhead to the software caused by excess type definitions is small. Another issue with the type aggregation is that added types are not removed from the aggregated server when a source server is

removed. However, similarly this does not lead to any significant disadvantages.

5.4 Evaluation of Instance Aggregation

The instance aggregation feature enables creating a categorized aggregated address space where information from different servers can be integrated under common entry points. It is able to work with any information model if configured so by the user. The aggregation procedure is flexible and can be configured for different use cases using the aggregation rules that utilize fairly complex logic. The use of FeatureIdentifiers brings efficiency to the aggregation rules by allowing one rule to apply to numerous instances and aggregated entry points. FeatureIdentifiers can alleviate the need to configure a large number of rules and values in complex aggregation scenarios. They enable the creation of aggregation rules even in cases where some of the characteristics of the aggregated information is not known during the configuration phase.

The implemented information model for the aggregated address space might not be suitable for every use case or might include unnecessary constructs. Many applications will only model, for example, physical assets, meaning the other categories of information are not needed. However, the information model serves mainly as a preliminary example and could be easily changed based on user feedback from real-life scenarios.

The aggregation procedure creates container Objects that integrate entry points for the different models, but more sophisticated aggregation methods that might integrate and transform the underlying model hierarchies could be possible directions for further development. Additionally, the aggregation could take into account overlappings in the different models by combining semantically similar information from different sources.

New source servers can be added during runtime and the aggregated instances are removed if the source server is removed. Nevertheless, the aggregation cannot react to runtime changes, such as additions of new Objects in the underlying servers. The aggregation process needs to be repeated from the beginning to accomodate such changes. The aggregation procedure is compatible with custom ReferenceTypes in the instance hierarchy tree as long as they are subtypes of HierarchicalReferences. On the other hand, AggregationRules can only utilize ReferenceTypes that are present in the OPC UA specification. These aspects could be improved upon in later development efforts.

Further development of the instance aggregation procedure could include more customization options. Currently, the AggregationFootprint only includes AND-based logical rules, but these could quite easily be extended with NOT-, XOR- and OR-logic rules. In addition, the customization of the aggregation parameters and especially the information model of the aggregated address space could be improved. The possibility to customize the names of the AggregateObjects could be a useful addition and easily added to the existing rule architecture.

The configuration of the aggregation rules and parameters could be performed using a graphical user interface. The GUI could improve usability greatly by ab-

stracting the underlying aggregation rule constructs to a more easily understandable form. This would make it possible to configure the rules without the need for special expertise. The GUI could employ some error-checking and limitations that decrease the chances for user-related errors in the configuration process. This graphical user interface feature would be implemented as an addition to the existing Historian Configurator software that already offers an GUI for modifying the parameters of the Historian software. Another important addition would be that the aggregation rules configured through the GUI would be persisted into a SQL database. In this manner, the aggregation rules could outlive the lifetime of the application process and could be loaded from the database later when the application is restarted. The database connection is already implemented in the Historian application but saving the aggregation rules to the database would require creating a suitable object-relational mapping.

6 Conclusions

OPC UA is a communication protocol created to meet the varied demands of industrial information systems with its flexible information modelling capabilities. It is expected to have wide use in the industry. With developments such as the Industrial Internet of Things and Industrie 4.0, the amount of data in the industrial environment is increasing and it is provided by an increasing number of sources. However, gaining a holistic view of system information from all the multiple heterogeneous providers can be difficult and inefficient. An aggregating OPC UA server is a paradigm used to create entry points to numerous source servers from a single OPC UA server. Nonetheless, no finished solutions exist for utilizing aggregating servers to integrate numerous sources of information implementing different information models and on different source servers. This thesis set out to improve the existing version of the aggregating OPC UA server in the Prosys OPC UA Historian software with the capability to aggregate such heterogeneous information sources.

First the information modelling characteristics of OPC UA were studied along with the companion specification information models. This knowledge, along with the needs of the perceived use cases for the new functionality, were used to create a set of requirements for the implementation. The implementation phase started by examination of the source code and software structure of the existing Historian software that utilizes Java programming language and the Prosys Java OPC UA SDK. The new aggregation functionality was then implemented as a new modular feature in the Historian application.

The implemented software feature consisted of three main parts: type aggregation, instance aggregation and service mappings. Firstly, type aggregation is responsible for accessing type information from configured source servers and copying it to the aggregating server. This is required by the aggregation procedure to keep types consistent across different instances from various source servers. Secondly, the instance aggregation integrates different information instances together in the aggregated address space of the OPC UA server in the Historian, thus creating a central access point to the different models. Aggregation rules were created to allow the user to configure how the instance aggregation is performed. In this manner, the aggregation procedure can meet various requirements in diverse use environments. Finally, new service mappings were created to combine the two previous elements by allowing the aggregated instances to make use of the common aggregated types on the Historian server.

The implemented solution manages to create a proof-of-concept prototype that meets the defined requirements. The aggregation feature is able to integrate disparate information from different information models and different source servers. The type definitions of these instances are kept consistent through the use of the type aggregation and the service mappings. The feature is a fairly generic framework that can be applied in a wide array of application areas and can be configured through the aggregation rules to be compatible with different systems and usage environments. The performance was found to be acceptable, even though it was not a priority during development.

Based on testing, the type aggregation procedure functions properly with all the companion specifications analyzed in this thesis and also with different custom information models. Information models are built as extensions of the base structure defined by the OPC UA specification and therefore they should be inherently compatible with each other. However, it is possible that a server exposes erroneous type definitions that will interfere with the type aggregation algorithm. A very realistic limitation on the type aggregation comes from different versions of information models. When various servers expose types that have same identifiers but different characteristics, they will cause issues for the type aggregation. In this case the type definition characteristics of instances can change between the source server and the aggregating server or it can even lead to faulty type definition semantics.

The instance aggregation also performed properly in the test cases, managing to integrate instances from both different servers and from different information models. The functionality of the instance aggregation can be flexibly configured so that it functions with different system platforms and information models. Nevertheless, it should be tested in real-life scenarios to discover how it meets actual user demands. Likely, more customization options are needed for the aggregation procedure and for the model of the aggregated address space.

This thesis does not offer an ultimate solution to the issues regarding information integration in OPC UA, but lays the path ahead for future solutions by offering the outlines of a prototype solution. This prototype will be used as the starting point for future incremental development in the Historian software. Before commercial release, the feature requires refining, testing and more functionality. Clear paths for future development are the addition of a fully-fledged graphical user interface for configuring the aggregation rules and parameters. The GUI should be made easy to use, so that even non-experienced users can control the aggregation process properly. The aggregation rules should be configured through the GUI to abstract the programming language–specific and intricate definitions to a more user-friendly form. In addition, the rules and configuration settings should be saved to a database so that they can be loaded when the program is restarted. Overall, more customization options and functionalities should be added in the future, possibly on the basis of feedback from test users.

References

[1] Mahnke, W., Leitner, S-H. and Damm, M., *OPC Unified Architecture.* Springer, 2009.

[2] OPC Foundation. *OPC Unified Architecture Specification, Part 1: Overview and Concepts.* Release 1.03, 2015.

[3] Großmann, D., Bregulla, M., Banerjee, S. and Schulz, D. *OPC UA Server Aggregation – The Foundation for an Internet of Portals.* Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA). Barcelona, Spain. 16-19. September, 2014.

[4] Großmann, D. and Banerjee, S. *Aggregation of Information Models – An OPC UA based approach to a Holistic Model of Models.* 2017 4th International Conference on Industrial Engineering and Applications (ICIEA). Nagoya, Japan. 21-23. April, 2017.

[5] Wollschlaeger, M., Fernbach, A., Kastner, W., Mätzler, S. and Huschke, M. *An OPC UA cross-domain Information Model for Energy Management in Automation Systems.* Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE. Vienna, Austria. 10-13. November, 2013.

[6] Wollschlaeger, M., Fernbach, A., Kastner, W. and Mätzler, S. *An OPC UA Information Model for Cross-Domain Vertical Integration in Automation Systems.* Emerging Technology and Factory Automation (ETFA), 2014 IEEE Proceedings of the 2014 IEEE Emerging Technology and Factory Automation. Barcelona, Spain. 16-19. September, 2014.

[7] Hästbacka, D., Barna, L., Karaila, M., Liang, Y., Tuominen, P. and Kuikka, S. *Device Status Information Service Architecture for Condition Monitoring Using OPC UA.* Emerging Technology and Factory Automation (ETFA), 2014 IEEE Proceedings of the 2014 IEEE Emerging Technology and Factory Automation. Barcelona, Spain. 16-19. September, 2014.

[8] Harju, J. *Plant Information Models for OPC UA: Case Copper Refinery.* Master's thesis, Tampere University of Technology, Tampere, 2015.

[9] Elovaara, J. *Aggregating OPC UA Server for Remote Access to Agricultural Work Machines.* Master's thesis, Aalto University School of Electrical Engineering, Espoo, 2015.

[10] Tuomi, I. *Aggregating OPC UA Server for Flexible Manufacturing Systems.* Master's thesis, Aalto University School of Electrical Engineering, Espoo, 2015.

[11] Tuovinen, T. *OPC UA Address Space Transformations.* Master's thesis, Aalto University School of Electrical Engineering, Espoo, 2016.

[12] Seilonen, I., Tuovinen, T., Elovaara, J., Tuomi, I. and Oksanen, T. *Aggregating OPC UA servers for monitoring manufacturing systems and mobile work machines.* 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA). Berlin, Germany. 6-9. September, 2016.

[13] OPC Foundation. *OPC Unified Architecture Specification, Part 3: Address Space Model.* Release 1.03, 2015.

[14] OPC Foundation. *OPC Unified Architecture Specification, Part 4: Services.* Release 1.03, 2015.

[15] OPC Foundation. *OPC Unified Architecture Specification, Part 5: Information Model.* Release 1.03, 2015.

[16] OPC Foundation. *OPC Unified Architecture for Devices Companion Specification.* Release 1.01, 2013.

[17] OPC Foundation. *OPC Unified Architecture for Analyser Devices Companion Specification.* Release 1.1a, 2015.

[18] OPC Foundation. *OPC Unified Architecture for ISA-95 Common Object Model Companion Specification.* Release 1.00, 2013.

[19] PLCopen and OPC Foundation. *OPC UA Information Model for IEC 61131-3.* Release 1.00, 2010.

[20] AutomationML e.V. and OPC Foundation. *OPC UA Information Model for AutomationML.* Release 1.00.00, 2016.

[21] Tina Lee, Y. *An Overview of Information Modeling for Manufacturing Systems Integration* National Institute of Standards and Technology, Manufacturing Systems Integration Division. Gaithersburg, Maryland, United States, 1999.

[22] Zhao, Y., Brown, R., Kramer, T. and Xu, X. *Information Modeling for Interoperable Dimensional Metrology.* Springer, 2011.

[23] Halpin, T. and Morgan, T. *Information Modeling and Relational Databases.* 2nd ed. Elsevier, 2008.

[24] Dahchour, M., Pirotte, A. and Zimanyi, E. *Generic Relationships in Information Modeling.* Journal On Data Semantics IV, 2005, vol. 3730, pp. 1-34.

[25] Weyer, S., Schmitt, M., Ohmer, M. and Gorecky, D. *Towards Industry 4.0 - Standardization as the crucial challenge for highly modular, multi-vendor production systems.* IFAC-PapersOnLine, 2015, vol. 48:3, pp. 579-584.

[26] Schleipen, M., Gilani, S.-S., Bischoff, T. and Pfrommer, J. *OPC UA & Industrie 4.0 - enabling technology with high diversity and variability.* Procedia CIRP, 2016, vol. 57, pp. 315-320.

[27] Brooks, S. *The Automation Pyramid: Soon to be Ancient History?.* [Online]. Available: https://www.tttech.com/news-events/tech-talk/details/the-automation-pyramid/. [Accessed on 10.10.2017]

[28] Asikainen, J. *OPC UA Java History Gateway with Inherent Database Integration.* Master's thesis, Aalto University School of Electrical Engineering, Espoo, 2014.

[29] Palonen, O. *Object-oriented implementation of OPC UA information models in Java.* Master's thesis, Aalto University School of Science and Technology, Espoo, 2010.

[30] OPC Foundation. *OPC Unified Architecture Specification, Part 8: Data Access.* Release 1.03, 2015.

[31] OPC Foundation. *OPC Unified Architecture Specification, Part 9: Alarms & Conditions.* Release 1.03, 2016.

[32] OPC Foundation. *OPC Unified Architecture Specification, Part 10: Programs.* Release 1.03, 2015.

[33] OPC Foundation. *OPC Unified Architecture Specification, Part 11: Historical Access.* Release 1.03, 2015.

[34] OPC Foundation. *OPC Unified Architecture Specification, Part 13: Aggregates.* Release 1.03, 2015.

[35] Pauker, F., Frühwirth, T., Kittl, B. and Kastner, W. *A Systematic Approach to OPC UA Information Model Design.* Procedia CIRP, 2016, vol. 57, pp. 321-326.

[36] International Society of Automation. *ANSI/ISA-95.00.02-2010 Enterprise/Control System Integration – Part 2: Object Model Attributes.*

[37] Karttunen, J. and Rossi, A. Outotec, Erbium. Rauhalanpuisto 9, 02230 Espoo. Interview, 6.9.2017.

[38] OPC Foundation. *OPC Unified Architecture Specification, Part 6: Mappings.* Release 1.03, 2015.